NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A TRUSTED PATH DESIGN AND IMPLEMENTATION FOR SECURITY ENHANCED LINUX

by

Allan T. Hilchie

September 2004

Thesis Advisor:	Cynthia E. Irvine
Thesis Co-Advisor:	David Shifflett

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2004	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: A Trusted Path Design and Implementation for Security Enhanced Linux			**5. FUNDING NUMBERS**
6. AUTHOR(S) Allan T. Hilchie			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The threat posed by malicious software and networked adversaries to computers has resulted in the development of mechanisms to provide assurance that security sensitive information is not being compromised. One such mechanism is called a Trusted Path. A Trusted Path provides a protected communications channel that permits the computer to authenticate itself to the user and for the user to authenticate to the system.

This thesis provides a demonstration implementation of a Trusted Path for Security Enhanced Linux (SELinux) and is used to examine trusted paths, their design and implementation. Additionally, the effectiveness of a Trusted Path for SELinux is analyzed.

This research is meant to provide a framework that could be used in combination with other efforts to enhance the security of SELinux.

14. SUBJECT TERMS Linux, Security Enhanced Linux, Trusted Path, Secure Attention Key, Computer Security			**15. NUMBER OF PAGES** 141
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release, distribution is unlimited

A TRUSTED PATH DESIGN AND IMPLEMENTATION FOR SECURITY ENHANCED LINUX

Allan T. Hilchie
Civilian, Department of the Navy
B.S., Northern Arizona University, 2000

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2004

Author: Allan T. Hilchie

Approved by: Cynthia E. Irvine
Thesis Advisor

David Shifflett
Thesis Co-Advisor

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The threat posed by malicious software and networked adversaries to computers has resulted in the development of mechanisms to provide assurance that security sensitive information is not being compromised. One such mechanism is called a Trusted Path. A Trusted Path provides a protected communications channel that permits the computer to authenticate itself to the user and for the user to authenticate to the system.

This thesis provides a demonstration implementation of a Trusted Path for Security Enhanced Linux (SELinux) and is used to examine trusted paths, their design and implementation. Additionally, the effectiveness of a Trusted Path for SELinux is analyzed.

This research is meant to provide a framework that could be used in combination with other efforts to enhance the security of SELinux.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
A. OVERVIEW 1
B. HISTORICAL BACKGROUND 1
C. PURPOSE OF STUDY 3
1. Scope and Assumptions 3
2. Research Objectives 3

II. OVERVIEW OF SECURITY ENHANCED LINUX 5
A. BACKGROUND 5
B. SELINUX SECURITY OVERVIEW 7
C. SELINUX SECURITY CONCEPTS 8
1. Type 8
2. Domain 8
3. Identity 8
4. Role 9
5. Security Context 9
6. Transition 9
D. SELINUX SECURITY ASSESSMENT 9

III. COMMON CRITERIA TRUSTED PATH REQUIREMENTS 11

IV. AN ABSTRACT TRUSTED PATH 15
A. ASSUMPTIONS 15
B. THREATS 16
1. Sniffing 16
2. Spoofing 16
3. Denial of Service 16
4. Circumvention of the Trusted Path 17
C. SYSTEM GOALS 17
D. SUBVERTING THE SYSTEM GOALS 18
E. DESIGN APPROACHES 19

V. DESIGN REQUIREMENTS FOR THE SECURITY ENHANCED LINUX TRUSTED PATH 21
A. COMMON CRITERIA REQUIREMENTS 21
1. FTP_TRP.1.1 21
2. FTP_TRP.1.2 21
3. FTP_TRP.1.3 21
4. Function Requirement Discussion 22
B. NON-FUNCTIONAL REQUIREMENTS 22
C. REFINING THE FUNCTIONAL REQUIREMENTS 22
1. Functional Sub-Requirements Supporting FTP_TRP.1.1 23
2. Functional Sub-Requirements Supporting FTP_TRP.1.2 23

3. Functional Sub-Requirements Supporting FTP_TRP.1.3 23
4. Additional Functional Sub-Requirements 23

VI. HIGH LEVEL DESIGN FOR THE SECURITY ENHANCED LINUX TRUSTED PATH 25
A. DESIGN REQUIREMENT DISCUSSION 25
B. ARCHITECTURAL REQUIREMENTS 25
1. Trusted Path Kernel Module (TP_Kern) 26
2. Trusted Path User Module (TP_User) 26
C. TRUSTED PATH STATES 26
1. Stopped 26
2. Awaiting SAK from User 26
3. Authentication 26
4. In the Trusted Path Menu 27
5. Normal Execution 27
D. INTERMODULE COMMUNICATION REQUIREMENTS 28
1. TP_Kern to TP_User 28
2. TP_User to TP_Kern 28
E. TP_KERN MODULE 29
1. Interfaces 29
2. Databases 29
3. TP_KERN External Interfaces 30
a. void handle_tp_sak() 30
b. asmlinkage int sys_trustedpath(int type_msg, int pid)) 31
c. REGISTER 31
d. SUSPEND 31
e. RESTORE 31
f. KILL 31
F. TP_USER MODULE 31
1. Interfaces 31
2. Databases 31
G. SUPPORTING MECHANISMS 32

VII. TRUSTED PATH IMPLEMENTATION 33
A. INTRODUCTION 33
B. CONSOLE TERMINOLOGY 33
C. INTERCEPTING OF THE SECURE ATTENTION KEY (SAK) 34
D. KERNEL MODULE OF THE TRUSTED PATH 35
E. CREATION OF A TRUSTED PATH PARENT ID 36
F. ADDITION OF A TRUSTED PATH SYSTEM CALL 36
G. REPLACEMENT OF THE GETTY AND LOGIN PROCESSES 37
H. PROCESS SUSPENSION AND RESTORATION 39
I. CONTROLLING THE TTY 39

VIII. TRUSTED PATH TEST PROCEDURES 41
A. INTRODUCTION 41
B. FSR 1 41

1. FSR 1 Purpose 41
2. FSR 1 Testing Rationale 41
3. FSR 1 Test 1 41
4. FSR 1 Test 2 41
5. FSR 1 Test 3 41
6. FSR 1 Test 4 41
7. FSR 1 Test 5 42
8. FSR 1 Test 6 42
9. FSR 1 Test 7 42
10. FSR 1 Test 8 42
11. FSR 1 Test 10 42
B. FSR 2 42
1. FSR 2 Purpose 42
2. FSR 2 Testing Rationale 42
3. FSR 2 Test 1 42
4. FSR 2 Test 2 43
5. FSR 2 Test 3 43
C. FSR 3 43
1. FSR 3 Purpose 43
2. FSR 3 Testing Rationale 43
3. FSR 3 Test 43
D. FSR 4 43
1. FSR 4 Purpose 43
2. FSR 4 Testing Rationale 43
3. FSR 4 Test 43
4. FSR 4 Test 2 43
5. FSR 4 Test 3 44
6. FSR 4 Test 4 44
7. FSR 4 Test 5 44
8. FSR 4 Test 6 44
9. FSR 4 Test 7 44
10. FSR 4 Test 8 44
E. FSR 5 44
1. FSR 5 Purpose 44
2. FSR 5 Testing Rationale 44
3. FSR 5 Test 1 44
F. FSR 6 45
1. FSR 6 Purpose 45
2. FSR 6 Testing Rationale 45
3. FSR 6 Test 1 45
4. FSR 6 Test 2 45
G. FSR 7 45
1. FSR 7 Purpose 45
2. FSR 7 Testing Rationale 45
3. FSR 7 Test 1 45

4. FSR 7 Test 2 45
H. FSR 8 45
1. FSR 8 Purpose 45
2. FSR 8 Testing Rationale 45
3. FSR 8 Test 1 46
4. FSR 8 Test 2 46
I. FSR 9 46
1. FSR 9 Purpose 46
2. FSR 9 Testing Rationale 46
3. FSR 9 Test 1 46
J. FSR 10 46
1. FSR 10 Purpose 46
2. FSR 10 Testing Rationale 46
3. FSR 10 Test 1 47
IX. CONCLUSIONS 49
A. ANALYSIS AND DISCUSSION 49
1. The Increasing Complexity of Software 49
2. Security Enhanced Linux 50
3. Adding Security to Existing Systems 51
B. FUTURE WORK 52
APPENDIX A. TRUSTED PATH –USER SPACE 53
APPENDIX B. TRUSTED PATH – KERNEL 75
APPENDIX C. TRUSTED PATH SYSTEM CALL 83
APPENDIX D. SCHEDULE MODIFICATION 85
APPENDIX E. KEYBOARD DRIVER MODIFICATIONS 95
APPENDIX F. FORK MODIFICATION 99
APPENDIX G. INSTALLATION GUIDE 107
APPENDIX H. SE LINUX POLICY CONFIGURATION 109
LIST OF REFERENCES 119
INITIAL DISTRIBUTION LIST 123

LIST OF FIGURES

Figure 1. Trusted Path States....28
Figure 2. Session Grouping by *tp_id.*....30

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. DUE-0114018.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. OVERVIEW

The widespread incorporation of computing systems into the everyday lives of average people has led to the realization that computers need to be 'secure'. There are numerous news articles which describe computer security concerns as well as the 'hack' of the week. Why computers need to be 'secure' can be readily described in terms of the need to protect information. Governments, businesses and individuals rely on information stored in computers to be correct, safe from eavesdropping and to be readily accessible. In computer security terms, the user needs assurance that his information stores support confidentiality, integrity and availability.

The need for computer security is not new but the requirement is increasing as more information is stored electronically and the number of interconnections between computer systems increases. As a small part of the general research in computer security subjects, this project will explore the implementation of a trusted path in Security Enhanced Linux (SELinux). This version of Linux is part of a National Security Agency (NSA) research project pursuing architectures for more secure operating systems. The addition of a trusted path to SELinux would increase the user's protection from password sniffing software and is a common security feature in medium and high assurance operating systems.

B. HISTORICAL BACKGROUND

Following quickly the development of general purpose computing machines, concerns about computer security arose. Initially the concerns were focused on non-malicious activities that could affect other computer users or that could cause inadvertant corruption of the operating system, but it soon became apparent that malicious behavior could be a threat. In 1970, a Rand report by Ware [1] described some of the potential computer security threats to the Department of Defense. The Ware report also made some general recommendations as to approaches to address the issues but it fell short of providing a concrete general purpose solution to the problems.

A general solution to the computer security problem was proposed in a paper by Anderson in 1972 [2]. Anderson's proposal was to create a reference monitor that would oversee all security related activities on the computer. Specifically, the reference monitor would ensure that all data accesses were correct in terms of permissions and actions. The reference monitor is generally accepted as an idealized mechanism to provide security in a 'high-assurance' computer system. Another related concept is the Trusted Computing Base (TCB). In the National Information Assurance Glossary [3], the TCB is defined as the

> [t]otality of protection mechanisms within a computer system, including hardware, firmware, and software, the combination responsible for enforcing a security policy.

The TCB consists of more than an implementation of a reference validation mechanism and includes such things as identification and authentication, the security administrator interface, audit retrieval and analysis functions [4].

A potential security problem with computer systems is that it is possible to write a computer program that can impersonate the login process or other security sensitive operation. This could allow the capture of a user's password and ID. This area of concern was identified as early as 1975 by Saltzer and Schroeder [5]. The idea of ensuring that a user was talking to the 'Trusted Computing Base' was required for some levels of computer assurance in the 'Orange' book [6].

> DoD 5200.28-STD: The TCB ("Trusted Computing Base") shall support a trusted communication path between itself and user for initial login and authentication. Communications via this path shall be initiated exclusively by a user.

This trusted communication path is generally termed a *trusted path* and is needed to give the user assurance that he is indeed communicating with the trusted computing base and not some software spoofing the TCB. Conversely, it also provides assurance that the operating system is connected to the user. The *trusted path* is important when the user identifies himself to the computer through the use of some sort of authentication process to avoid the malicious capture of the user's authentication information.

C. PURPOSE OF STUDY

1. Scope and Assumptions

This thesis will assess the requirements for a Trusted Path in Security Enhanced Linux. The publicly available information about the various Trusted Path implementations will be reviewed in an attempt to identify significant design considerations. After reviewing previous work in this area, a design for a Security Enhanced Linux implementation of a Trusted Path will be proposed and implemented.

2. Research Objectives

The Information Assurance Research Office of the National Security Agency (NSA) has worked with various research organizations to develop Security Enhanced Linux [13]. One purpose of NSA's research is to provide a demonstration project of how mandatory access controls (MAC) can be used to confine processes to prevent security issues. The project is also working on developing a security implementation architecture that is flexible enough to allow for the implementation of various security policies [17].

The research presented in this paper is meant to provide a design document for Trusted Path implementation using the Security Enhanced Linux as a demonstration project.

THIS PAGE INTENTIONALLY LEFT BLANK

II. OVERVIEW OF SECURITY ENHANCED LINUX

A. BACKGROUND

Information assurance is a very active concern of the U.S. government, especially in the agencies and departments that deal with highly classified information. It seems reasonable that Top Secret information stored on computers should be protected from unauthorized disclosure, e.g. protected from computer hacking. The National Security Agency (NSA) is concerned with systems that are capable of protecting the agency's sensitive information. As a result, the NSA conducts research in improving computer security, and operating system security is an area of particular interest.

One of the current computer security debates is whether "open source" software is more secure than proprietary software. This discussion pertains to the relative security of each of the software models. The Linux operating system is the most commonly identified open source software in these discussions and is identified as open source because the source code is freely available. Most of the development of Linux is performed by various volunteers throughout the world. Proponents of open source believe that the availability of the source code for review by all users improves security. These users provide a potentially large number of reviewers to identify and correct some of the potential vulnerabilities. Others believe that the availability of the source code allows users to modify it to meet their particular security needs. Another potential advantage for open source software is the ability of the multitude of developers to react quickly to discovered vulnerabilities by providing software patches.

At the other end of the spectrum is proprietary or closed source code software, such as the Microsoft operating system. Proponents of these types of products believe that open source products provide the potential hacker with an advantage in finding potentially exploitable vulnerabilities. They also cite the responsiveness of the commercial software industry as a key factor in providing the needed security.

An argument can also be made that by controlling the software from design through implementation to final delivery gives the commercial software company the ability to ensure security of the product. While this control of the entire product could help ensure the security of the system, in practice, it does not appear to function that way. Many commercial products have had additional software functionality, commonly referred to as 'Easter Eggs', embedded in the product by the implementation team. Some Easter eggs have included significant amounts of code such as the flight simulator found in an older version of Excel. Additional information about this and other 'Easter Eggs' is available at the Easter Egg Archive [28]. If the implementers can include an unauthorized flight simulator what prevents the addition of more malicious functionality by a member of the team? Witten, et al., [21] provides an excellent discussion of some aspects of the subject.

Ultimately, open vs. closed source is not particularly relevant to medium or high assurance software as defined by the U.S. Government. Open and closed are really economic models that describe how groups generate profit from the software. Assurance or computer security could potentially be provided under either model. What provides the needed assurance is the rigor of the development methodology. Without a formal and methodical approach to the design, implementation, testing, configuration management, etc., software will continue to have large numbers of bugs, including those that are security related, regardless of the economic model of the software. Generically, computer systems that have undergone a formal security design process can be termed trusted systems because of the assurance provided by the process.

Trusted computer systems have many characteristics that make them distinct from commonly used systems [2]. The lack of mandatory access controls (MAC) was identified as functionality absent from most 'mainstream' operating systems. As a result, NSA developed a MAC architecture that could be used for many current operating systems and flexible enough to allow a wide range of security policies to be applied. This architecture is known as Flask and Security Enhanced Linux (SELinux) is a research prototype of the Flask architecture [12].

Flask separates policy from enforcement as a means of increasing the flexibility of the system. NSA had a generic security policy built by NAI labs to support the SELinux demonstration project. SELinux provides some support for multilevel security but the implementation is incomplete at this time and is not used regularly within the SELinux user group. The selection of Linux as a demonstration platform for Flask was driven by the open source nature of the ongoing development and the desire to improve the security of a widely used platform [12].

B. SELINUX SECURITY OVERVIEW

The remainder of this section will address the specifics of the Flask demonstration implementation and its generic security policy. The following is based on the version of SELinux for the Linux 2.6 kernel. This kernel adds support for extended security attributes and made significant changes to the previous SELinux implementations.

SELinux can be viewed as having two primary components: a security policy and the enforcement mechanism. Policy is contained in the security server and enforcement is performed by object managers.

The security server is responsible for the security policy and for making determinations about security access. This server is located in the kernel as a subsystem and provides application program interfaces (API) that the object managers can query for security determinations. For example, if a user attempts to access a file, the file system object manager will submit the user credentials and the file attributes to the security server to determine if the user has the authority to access the file in the manner requested. The server is also responsible for security of the security policy.

The various object managers enforce the policy decisions of the security server. Object managers include kernel subsystems such as process management, file system, and socket IPC. When a process attempts to access an object for the first time, the object manager contacts the security server with the security contexts. The security server performs a look up to determine if a specific access is allowed and returns a decision to the calling object manager. The object managers may cache the decision in an access vector cache (AVC) in order to speed up recurring accesses. There is a mechanism to allow revocation of access permissions if the policy changes while an object is open.

This revocation mechanism is described as the most complex aspect of the Flask architecture in [10] and requires additional communication between the object managers and the security server to allow updating of policy settings even if an object is open.

C. SELINUX SECURITY CONCEPTS

1. Type

Each object, such as directories, files, and sockets, is assigned a type. An access matrix is created that contains allowable accesses between types. An access must be explicitly defined as allowable per the SELinux implementation of principle of least privilege. There are more than 140 different types in the default SELinux installation and they are grouped into seven categories:

- Device
- Devpts –pseudo terminal devices
- File
- Network
- Network File System
- Procfs – process context file system
- Security

2. Domain

Each process runs in a domain. The domain controls what the process can access. It is very much like the idea of types for objects and is stored in the same data structure as the types. There are in excess of 160 domains in a default install of SELinux including admin, user, auth-net, fcron, kernel, startx, and more than 150 various program domains.

3. Identity

Each user has a SELinux identity in addition to a standard Linux user ID. This can be confusing because the identity and ID are typically the same name. Normal commands that can change a Linux ID, e.g. su, do not change the SELinux identity.

Example:

Root performs an id command

```
root@localhost selinux]# id
uid=0(root) gid=0(root) groups=0(root), 1(bin), 2(daemon),
3(sys),          4(adm),          6(disk),          10(wheel)
context=root:sysadm_r:sysadm_t
```

Root su's to ahilchie

```
[root@localhost selinux]# su - ahilchie
```

Root performs an id command as ahilchie

```
[ahilchie@localhost ahilchie]$ id
uid=500(ahilchie)   gid=500(ahilchie)   groups=500(ahilchie)
context=root:sysadm_r:sysadm_t
```

In the example above the *uid*, *gid*, and *groups* are the Linux identity while the *context* is SELinux identity information.

4. Role

A role is a collection of domains that some grouping of users can enter. Currently, the domains that a particular user can enter are identified in various configuration files.

5. Security Context

A security context consists of an identity, a role and a domain or type.

Example contexts:

	identity:role:domain or type
Init process	system_u:system_r:kernel_t
Root	root:sysadm_r:sysadm_t
User	ahilchie: user_r:user_t

6. Transition

Transition decisions are made to determine the correct security context for a particular operation. As an example, if a user (ahilchie) creates a new file in his home directory, it will have a security context of ahilchie:object_r:user_home_t but the same file in the /tmp directory would be ahilchie:object_r:user_tmp_t.

D. SELINUX SECURITY ASSESSMENT

SELinux is not a high assurance system. It is Linux with additional security policies. There has been no attempt to correct any deficiencies in the underlying operating system. As noted on the NSA SELinux web page:

> This work is not intended as a complete security solution for Linux. Security-enhanced Linux is not an attempt to correct any flaws that may currently exist in Linux. Instead, it is simply an example of how mandatory access controls that can confine the actions of any process, including a superuser process, can be added into Linux. The focus of this work has not been on system assurance or other security features such as security auditing, although these elements are also important for a secure system. [17]

Security system subversion remains a very real possibility. It would be possible to create a program and add it to a Linux distribution. Properly coded, the added program could provide any desirable level of access to the system. As an example of the potential, consider Karger and Schell's suggestion of a compiler trapdoor [20] which was elaborated by Ken Thompson in his discussion of UNIX system security [15]. For a discussion of the threat of subversion see Anderson, et al [19]. The inclusion of programs in the Linux source tree is not tightly controlled nor are the programs rigorously examined (in a formal methods manner) prior to being added.

A more simplistic attack vector would be to use some current exploit against the Linux kernel. These exploits are uncovered regularly and, depending on the properties of the exploit, might allow a malicious user to subvert the SELinux MAC policy and take control of the system.

Security Enhanced Linux is a fairly complex overlay to the Linux operating system and contains in excess of 350 configuration files. The Linux system is also a complex system. Complexity is a serious challenge to security for a number of reasons [2]. This complexity presents very real challenges to administration of the system.

III. COMMON CRITERIA TRUSTED PATH REQUIREMENTS

Traditionally, trusted computing systems have been predominately used by governmental organizations[1]. This chapter will look at the U.S. Government requirements for a *trusted path* in these types of systems. This review of governmental standards will provide a framework for the demonstration implementation of the Security Enhanced Linux trusted path. There has been considerable effort made by many people to identify and codify various aspects of assurance that can be leveraged in this project.

The U.S. Government and many other nations have adopted the Common Criteria as the security standard for computing systems. The Common Criteria identifies a Security Functional requirement class, 'FTP: Trusted Path/channels' [7]. The Trusted Path class identifies several requirements for a trusted path including:

> Trusted path requires that a trusted path between the TSF[2] and a user be provided for a set of events defined by a PP/ST author. The user and/or the TSF may have the ability to initiate the trusted path.
>
> The TSF shall provide a communication path between itself and local users that is logically distinct from other communication paths and provides assured identification of its end points and protection of the communicated data from modification or disclosure.
>
> The TSF shall require the use of the trusted path for initial user authentication.

In other words, the trusted path must be an unforgeable communication mechanism between the user and the TCB. Once the user initiates the trusted path, he can be certain that the TCB has taken control and as long as the user remains in the trusted path, then, there is confidence that any software trying to spoof the TCB is thwarted.

---

[1] Evaluated systems have been predominately used by governments; it could be effectively argued that commercial enterprises should have a significant interest in these systems as well. For the government, the compromise of sensitive information could have dire consequences. The same argument is applicable for businesses. The compromise of some types of sensitive business information could even result in the complete failure of the business.

[2] Target of Evaluation Security Functions – Parts of the system that have to be relied upon for enforcing security policies

The Common Criteria allows for the TSF to initiate a trusted path. This capability on the part of the TSF would either degrade the level of security and/or significantly increase the complexity of the TCB. Typically the trusted path is entered into by the system reacting to some user action. The TCB reacts to the user input and provides assurance that the communications between the user and the TCB are protected. If the system is allowed to initiate a trusted path, the user would be unable to determine if the communications path is to the TCB or to a spoofing program. This implies that the TCB would have the responsibility to prevent any system output that appears to be a trusted path. While this may be possible, it would increase the complexity of the TCB's responsibilities with regard to system output and therefore, reduce the system security.

The National Information Assurance Partnership (NIAP) is an affiliation between the NSA and Common Criteria Vendors. NIAP maintains a Public Interpretations Database [8] of determinations made by the Interpretation Board. These are meant to provide additional details of the security requirements required under the Common Criteria. The NIAP Interpretation I-0191: *Minimum Set Of Actions That Require Trusted Path* identifies the following required functions:

Authentication.

Session establishment where identification is provided.

Requests to change the subject sensitivity label.

Requests to display the current subject's complete sensitivity label.

Requests to assume a security administrative role.

This list identifies a set of actions which if implemented by the operating system must use the trusted path mechanism. The common thread in the listed items is that they have some security sensitive function. In the case of authentication, the user is providing some sensitive user identifying information (typically a password) that must be protected from entities outside of the TCB. Performing security sensitive actions on a system also requires that the administrator ensure that he is dealing with the TCB prior to issuing security relevant commands.

The sensitivity label requirements are for multilevel systems. These are computer systems that are used to process information at different levels of sensitivity and where there is a requirement to prevent information 'leakage' between different sensitivity levels. A concrete example of a multilevel secure system would be a single computer where both Top Secret and Secret information is processed. It is important to ensure that no Top Secret information can be accessed by a non-Top Secret process. If some software at the Secret level was able to spoof being a Top Secret process, it is possible that information could be compromised. The trusted path provides the authorized user with the ability to select the appropriate level for a session.

Another aspect of the Common Criteria framework are protection profiles. A protection profile is essentially a high level, abstract statement of system security requirements. An organization, such as a government, could create a protection profile, thus stating its security requirements. In this situation a user might be a specific government agency with specific requirements for security. This agency would then develop a protection profile that identifies all of its security requirements. Vendors would then be able to develop products that meet the protection profile requirements. For this project, there is no protection profile identified so we assume that the desire is to have a medium assurance system and design the trusted path to meet an evaluated assurance level 4 (EAL4) requirements.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. AN ABSTRACT TRUSTED PATH

The previous chapter defines the basic requirements of a trusted path. The primary requirement was that the trusted path must provide an unforgeable communication path between the user and the trusted system. This path is invoked by the user and must be evoked under certain conditions.

Open source documentation of trusted path design is limited in availability, therefore, this section will attempt to explore the design considerations for an abstract trusted path. "Abstract" is used to mean that the trusted path design considerations are removed from the actual details of the operating system and the computer hardware.

A. ASSUMPTIONS

This abstract trusted path will be confined to analysis of the trusted path as a local problem, i.e., the users will be at the system's local terminal. This paper will not address the more complex issue of a network trusted path. This project will also focus on a single user system in a single level security environment. In the design, we will attempt to allow the basic structure to be extendable to multi-user multilevel environments but these features are not part of the project.

It is assumed that the security policy requires that each user must authenticate to the system through the trusted path for each and every computer session.

A trusted computing base (TCB) is assumed to exist and is not corruptible. The evaluation assurance level of the TCB must be commensurate with that of the trusted path and vice versa. An implementation of a trusted path might be likened to installing very secure external doors in a house. These external doors might be impregnable but the security of the house is also dependent upon preventing unauthorized access from windows, walls, etc., as well. The addition of a trusted path to Security Enhanced Linux will not correct any potential pre-existing vulnerabilities in this operating system.

B. THREATS

Any security relevant design should be based on a 'threat model'. This means that potential threats to correct functioning should be identified in order to develop strategies to counter those threats. A human metaphor will be used to develop the potential threats.

Alice (our user) has a secret (her password) that she wants to give Bob (our TCB). Crafty Hacker is the enemy and he wants to steal Alice's secret or, in some cases, keep Bob from getting it. Alice knows Bob and so her trusted path is achieved by the physical recognition of Bob. How can Crafty accomplish his goals?

He could lurk around the corner and eavesdrop on the conversation.

He could pretend to be Bob (perhaps he is a master makeup artist).

If these two approaches fail, he could then prevent Alice and Bob from meeting in a secure manner (standing between them)?

In the computer security parlance:

1. Sniffing

Essentially this is eavesdropping on the conversation. In our trusted path scenario this would require modifying some of the software used for the trusted path to allow redirection of the data.

2. Spoofing

This is impersonating some portion of the trusted path or TCB to allow capture of the sensitive data. Spoofing the logon process with a fake logon screen that accepts the user logon information, throws an error, and then starts the real logon process would be the classic example. A more sophisticated approach would be a man-in-the-middle attack where the spoofing software pretends to be the user to the TCB and the TCB to the user.

3. Denial of Service

DOS is the preventing of transmission of information. A process could be created with the purpose of preventing valid users or the TCB from accessing the trusted path.

4. Circumvention of the Trusted Path

There exists one other threat to our trusted path: circumvention of the trusted path. This circumvention is described as system subversion and is defined as '… the covert and methodical undermining of internal and external controls over a system lifetime to allow unauthorized or undetected access to system resources and/or information.' [18] The extent of the subversion threat is discussed in Anderson et al 2004 [19]. The subversion problem could be exemplified by a backdoor being installed in the system in order to avoid having to use the trusted path.

The high level system goals will be presented and then the methodology to prevent the threats will be developed.

C. SYSTEM GOALS

A user needs assurance that he is communicating with the trusted computing base prior to performing security sensitive operations. Examples of security sensitive operations include authentication, changing passwords and performing some administrative actions. The special case of authenticating to the trusted computer base will always occur at the beginning of a session and precedes other trusted path operations. Security sensitive operations will occur after the user is authenticated and while in the trusted path. Also, we may want to support authentication as a different role during an established session, e.g., su to root in the Unix sense.

How does the user invoke the trusted path? Any input methodology could potentially be made to work. This could be as simple as a keystroke combination or as complex as the use of a smart card inserted into a system card reader. Many current computer systems use either a keystroke combination, the mouse position or a click on the screen. Examples of the keystroke approach are found in the XTS 400 system [23] or the Windows 2000 operating system [24]. Trusted Solaris systems use the position of the mouse [25]. It would even be reasonable to have a separate button on the computer to invoke the trusted path. Generically the method to invoke a trusted path is referred to as a secure attention key (SAK).

How does the user know that the trusted path is in effect? Once again there is a range of potential ways to inform the user that the trusted path is in effect. The only real limit on this is the output capability of modern computers. Approaches could include some special LED on the computer or computer monitor that would only be on if the trusted path was in effect. Trusted Solaris has a special area on the screen that can not be affected by non-TCB processes. A simple approach is to guarantee that pressing the SAK always invokes the trusted path. This appears to be the approach used by the XTS 400 and in the Windows NT family. The TCB must be able to protect the SAK invocation; in other words, once the SAK is initiated, no process should be able to prevent that signal from reaching the TCB. This will prevent some process from intercepting the SAK and presenting a spoofed trusted path.

Once the trusted path is invoked, the input and output from the trusted path must be protected from any and all other un-trusted processes. Conceptually this could be achieved by approaches such as terminating or suspending any running un-trusted process. Another approach would be to create a separate and isolated process for use by the trusted path; the major concern in this approach would be to prevent the other processes from gaining access to the terminal.

D. SUBVERTING THE SYSTEM GOALS

Sniffing and spoofing threats can be avoided by careful design. Toward that end, there are two design facets that need to be addressed. First, that the SAK must always be controlled by the TCB and second, the TCB must block any input and output from untrusted software while the trusted path is in effect. The combination of these two design facets will prevent both sniffing and spoofing of the trusted path communication.

Preventing denial of service (DOS) is more difficult because the problem is larger than the trusted path. There is no way for the trusted path to prevent the majority of DOS attacks. A type of attack that the trusted path would be unable to prevent is unplugging the computer. In terms of the trusted path, what types of DOS attacks should be prevented? Any set of non-TCB processes should not be able to prevent a user from invoking the trusted path. This may be achievable within the TCB and requires that the

TCB process keyboard interrupts correctly. If the TCB can guarantee that the SAK generates an interrupt, which can be used to start the trusted path, then the problem is adequately solved. This does not solve all potential denial of service issues.

The last threat is the circumvention issue. This project will not attempt to address all manner of system subversion. Only a high assurance system, such as those constructed to meet EAL 6 and EAL 7 criteria, can provide any confidence that the system has not been subverted

E. DESIGN APPROACHES

A primary design goal is simplicity. Saltzer and Schroeder identified complexity as a major security issue in their seminal work in 1975 [5]. This implementation will attempt to minimize complexity unless doing so will reduce security. Most of the high level design decisions are made based on this goal. These include:

- The trusted path shall be activated by a keystroke combination (SAK).
- Pressing the SAK shall always invoke the trusted path.
- The software implementation of the trusted path should be as small as possible.
- The trusted path mechanism will be as simple as possible. For example, the trusted path will not use a 'windows style' graphical display due to the large amount of code that would have to be trusted.
- The other running processes will be isolated from the trusted path to prevent them from being able to communicate with the trusted path, keyboard or the computer monitor.
- The authentication mechanism shall only accept input from the trusted path.

THIS PAGE INTENTIONALLY LEFT BLANK

V. DESIGN REQUIREMENTS FOR THE SECURITY ENHANCED LINUX TRUSTED PATH

The Common Criteria defines three functional requirements for a medium assurance trusted path. The preceding abstract trusted path discussion identified some additional design goals. This chapter will present the specific design requirements that were used in this project to design the SELinux trusted path.

A. COMMON CRITERIA REQUIREMENTS

The Common Criteria Security functional requirements document [22] provides a template that allows system designers to select from various phrases to build a Common Criteria protection profile. As an example, the following is the first Common Criteria requirement for a trusted path:

> FTP_TRP.1.1 The TSF shall provide a communication path between itself and [selection:remote, local] users that is logically distinct from other communication paths and provides assured identification of its end points and protection of the communicated data from modification or disclosure.

Based on the requirements for this demonstration implementation the 'local' phrase is selected to generate the following statement. Similarly the two other Common Criteria trusted path requirements are chosen to create our protection profile statements:

1. FTP_TRP.1.1

The TSF shall provide a communication path between itself and local users that is logically distinct from other communication paths and provides assured identification of its end points and protection of the communicated data from modification or disclosure.

2. FTP_TRP.1.2

For FTP 1.2 the selection was: The TSF shall permit local users to initiate communication via the trusted path.

3. FTP_TRP.1.3

For FTP 1.3 the selection was: The TSF shall require the use of the trusted path for initial user authentication.

4. Function Requirement Discussion

FTP_TRP.1.1 really defines a trusted path. The requirement states that the trusted path must provide an unforgeable communications link between the user and the trusted portion of the operating system. The trusted path must also ensure that, once established, any information passed between the user and the TSF must be protected from modification and disclosure.

FTP_TRP.1.2 requires that a user must be able to invoke the trusted path.

FTP_TRP.1.3 states that initial user authentication shall be conducted using the trusted path.

B. NON-FUNCTIONAL REQUIREMENTS

In addition to the functional requirements there are guiding design principles that are important to designing security features. Primarily these principles include simplicity, modularity and appropriate layering of the design.

Simplicity is a very important design consideration which enhances the understandability of the code. Increasing complexity increases the probability that serious design flaws will exist and decreases the chance that these flaws will be discovered during design review. Additionally, increased complexity leads to an increase in the likelihood of implementation mistakes.

Modularity and layering are related to simplicity in terms of design. Designing the modules and layers to group similar or related behavior enhance the understandability of the design. Placing functionality at the appropriate layer provides the same benefit.

C. REFINING THE FUNCTIONAL REQUIREMENTS

The SELinux trusted path has three functional requirements plus several non-functional requirements. Based on the proposed design, this section defines how this implementation meets the common criteria functional requirements for a trusted path using functional supporting requirements (FSR). These supporting requirements are simply the elements that this design needs to be able to meet the common criteria requirements.

1. Functional Sub-Requirements Supporting FTP_TRP.1.1

FSR 1 – The TSF provides a trusted path menu to allow the user to select the appropriate action to be performed.

FSR 2 – The TSF suspends all non-trusted path user processes associated with the user upon entering the trusted path.

FSR 3 – The communication path between the keyboard or computer monitor and the TSF shall be under control of the TSF while the trusted path is active.

2. Functional Sub-Requirements Supporting FTP_TRP.1.2

FSR 4 – The TSF is always notified of the secure attention key (SAK) signal.

3. Functional Sub-Requirements Supporting FTP_TRP.1.3

FSR 5 – User authentication is controlled via the trusted path.

FSR 6 – Pressing the SAK when there is no authenticated user results in the TSF initiating an authentication sequence.

FSR 7 – Failed authentication results in returning to the initial pre-authentication state.

FSR 8 – Successful authentication results in the display of the trusted path menu.

4. Additional Functional Sub-Requirements

The following two sub-requirements are necessary to provide appropriate functionality to the SELinux trusted path.

FSR 9 – Selecting *run* from the trusted path menu restores any previously suspended user processes or causes a new shell process to be spawned.

FSR 10 – Selecting *exit* from the trusted path menu terminates any user processes and returns the computer to the pre-authentication state.

These 10 functional supporting requirements are sufficient to meet the Common Criteria functional requirements, and support some additional desirable functionality as well.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. HIGH LEVEL DESIGN FOR THE SECURITY ENHANCED LINUX TRUSTED PATH

Previous chapters discussed functional requirements for the implementation of a trusted path. This chapter will begin with a discussion of some of the key requirements for the trusted path design. These requirements will result in architectural requirements for the design. The trusted path as a state machine will also be introduced. The final section of this chapter presents a set trusted path interfaces to be implemented in order to meet the functional requirements.

A. DESIGN REQUIREMENT DISCUSSION

To prevent some untrusted process from blocking the TCB from receiving the SAK the keyboard scancode should be captured at a low level within the system. This capture should occur inside of the kernel. By keeping the SAK processing in the kernel we can prevent user space processes from affecting the SAK.

Blocking any input/output by untrusted processes while in the trusted path can be accomplished in a number of ways including destroying or suspending any running untrusted processes. Another approach is to isolate the terminal so that only the trusted path can access it. This thesis will suspend user processes to prevent potential spoofing or sniffing.

Ensuring that the authentication process only receives input from the TCB can be accomplished by modifying the traditional *getty* and *login* processes. Both *getty* and the *login* processes currently run in user space.

B. ARCHITECTURAL REQUIREMENTS

The design requirements call for processing in both the kernel and in user space. This drives us toward two separate code modules. One module will be part of the kernel and will be responsible for kernel-related trusted path processing. The other module will be in user space. The functions performed by each module are listed as follows.

1. Trusted Path Kernel Module (TP_Kern)

- Catch the SAK
- Suspend user processes
- Restore user processes

2. Trusted Path User Module (TP_User)

- Maintain trusted path state
- React to TP_Kern SAK pressed message
- Control the TTY
- Display initial press SAK message
- Display TP menu
- React to input from TP menu
- Control user authentication
- Start a command shell
- Initialize TTY

C. TRUSTED PATH STATES

From the perspective of a trusted path we can view the computer as a state machine with several distinct states.

1. Stopped

In the *stopped* state the system is powered down and unavailable for interaction. It will transition to the next state as a result of the boot process. In and of itself this state is not interesting in the context of the trusted path.

2. Awaiting SAK from User

In the *no_user* state there are no user processes running on the machine but all of the necessary boot processes have run. The computer is waiting for a user to begin an authentication process. This state will be entered after the system boot process is complete. This state is also entered when a user selects exit to end his session with the computer or when a user incorrectly authenticates.

3. Authentication

In the *in_auth* state, the user is proving his identity to the TCB. There are several distinct steps to the authentication but from the perspective of the trusted path it may be

viewed as a single state. Successful completion of the authentication process results in moving to the trusted path menu. Unsuccessful completion results in returning to the *no_user* state.

4. In the Trusted Path Menu

In the *in_TP_menu* state the user has been authenticated and the trusted path is awaiting user input on his desired action. All user processes are suspended. There are two entry points; this state can be reached by successful completion of the authentication process or by pressing the SAK during the normal execution state. There are also two exit points, the user may exit the trusted path menu to the *user_run* state or to the *no_user* state. While in the *in_TP_menu* state, the user is authenticated to the system and the trusted path menu is displayed. The trusted path has control of the computer monitor and keyboard.

5. Normal Execution

The *user_run* state is the condition in which users may run applications and processes. The user is authenticated to the system and the trusted path is idle.

Figure 1 presents a pictorial representation of the trusted path states and the transitions between the states.

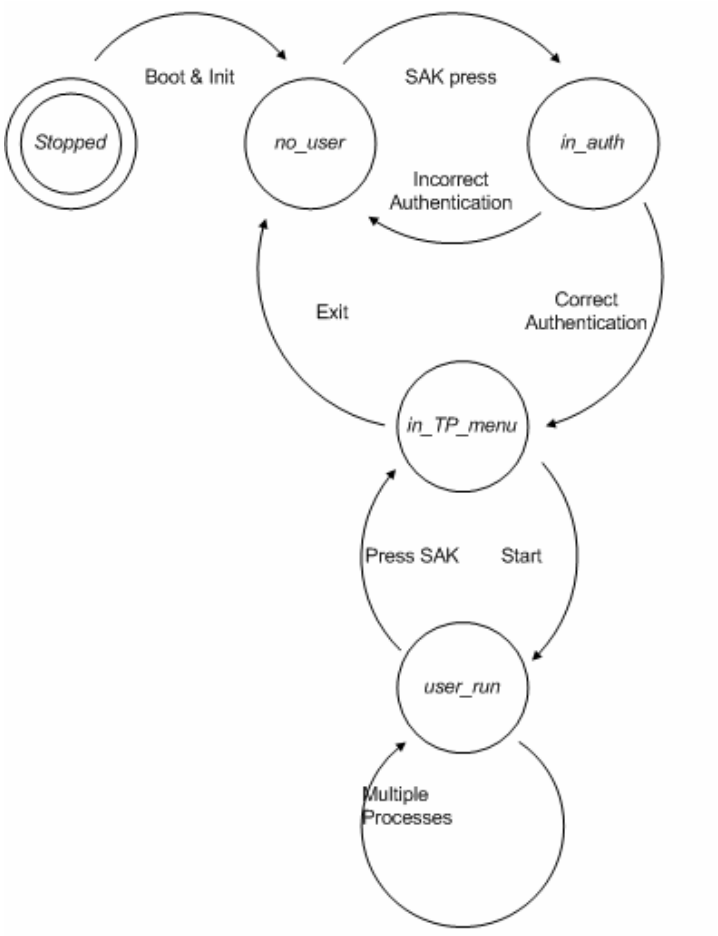

Figure 1. Trusted Path States

D. INTERMODULE COMMUNICATION REQUIREMENTS

1. TP_Kern to TP_User

Notify Trusted Path User Module that the SAK has been pressed

2. TP_User to TP_Kern

Four distinct messages must be sent from the user space trusted path module (UTPM) to the kernel trusted path module (KTPM):

- Register the process ID (PID) of the UTPM with the KTPM. With this information the KTPM can ensure that only the UTPM is making calls to the KTPM. It also will allow the KTPM to signal the UTPM.
- Suspend the processes associated with a particular trusted path ID (TP_ID). The TP_ID is the PID of the first user process in the parentage tree of the any given user process.

- Restore the processes associated with a TP_ID. This is used to restore any previously running processes on returning to a running state from the trusted path menu.
- Kill the processes associated with a TP_ID. This is used to destroy any user processes when the user exits the system.

E. TP_KERN MODULE

TP_Kern is the kernel space grouping of functionality for the trusted path. It is coded in the trustedpath.c as well as the trustedpath.h files.

1. Interfaces

void handle_tp_sak ()
asmlinkage int sys_trustedpath(int type_msg, int pid)

The interface was named with a handle prefix because this naming convention is commonly used in the Linux kernel development community.

2. Databases

An integer variable, labeled *tp_id* for trusted path ID, was created in the kernel schedule header file. This variable is used by the fork process to label each process with its *original parent* process ID. In this context, *original parent* is used to describe the first shell process. The *init* process is the only user space process started by the kernel and can be considered the original process in any process tree. The *tp_id* captures the session level after *tp_getty* and allows the trusted path to determine which processes are associated with a user session. The *tp_id* is used to suspend or terminate specific session processes as needed. Detailed information about these modifications are presented in Appendix D, Schedule Header Modification, and Appendix F, Fork Modification. Figure 2 presents a graphic representation of process hierarchy. The *tp_id* is associated with each session and is derived from the *PID* of the original shell for each session.

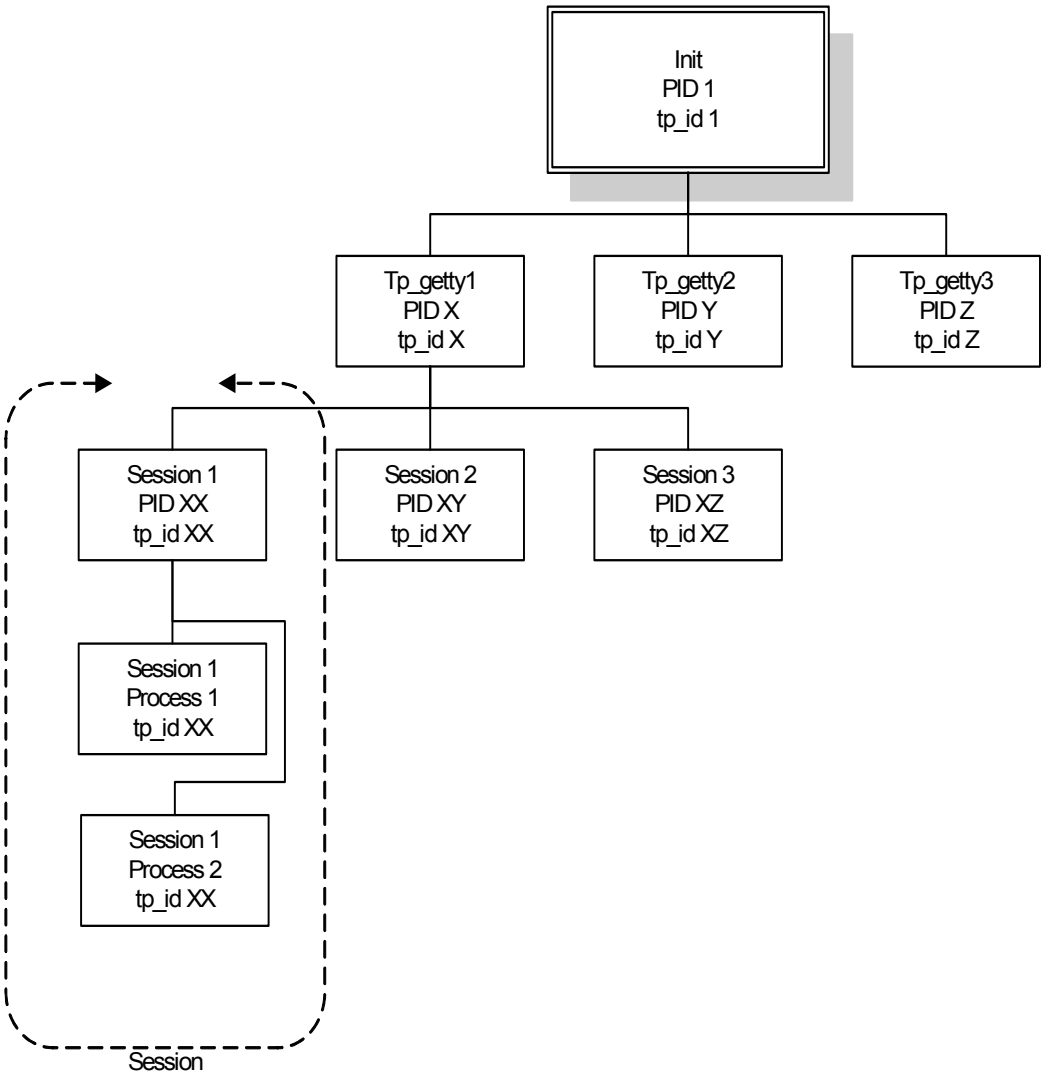

Figure 2. Session Grouping by *tp_id.*

3. TP_KERN External Interfaces

The following functions are implemented inside of the trusted path kernel module.

a. void handle_tp_sak()

Called by the keyboard driver in response to simultaneous pressing of the control and break keys. TP_Kern responds to the SAK by notifying the trusted path user space module with a signal.

b. asmlinkage int sys_trustedpath(int type_msg, int pid))

This trusted path system call is the user space interface that is used by TP_User to send information to the TP_Kern. This function returns a zero on successful completion. There are four messages that were identified as being necessary:

c. REGISTER

Called by the trusted path user module on initialization to pass the trusted path process ID to the TP_Kern. This information is stored in a kernel variable. This variable is used to allow the TP_Kern to signal the trusted path process.

d. SUSPEND

Called by TP_User to suspend all processes with the same tp_id as the PID that was passed to the function. This is done by suspending each process with a tp_id that is equal to the tp_id of the passed in PID. The processes are suspended by removing them from the run queue of the kernel's scheduling process.

e. RESTORE

Called by TP_User to restore all processes with the same tp_id as the PID. This is done by determining the tp_id of the PID. Then each process with this tp_id is restored by moving the process on the run queue of the kernel.

f. KILL

Called by TP_User in response to the user ending a session. TP_kern will then enumerate all the processes associated with the PID's tp_id and then issue a *force_sig(SIGKILL,pid)* to each associated PID as defined in signal.c.

F. TP_USER MODULE

The TP_User module is the user space grouping of functions for the trusted path. It is coded in the *login.c* file which is used to create the *tp_getty* executable file.

1. Interfaces

There are no external interfaces in the traditional sense. The only external input is notification of a SAK from the kernel. This is handled by use of a signal handler in the *tp_getty* executable.

2. Databases

The primary databases to support the trusted path implementation are a state variable and session structure.

The trusted path state variable is labeled *tp_state* and there are four valid states identified. These are the same states as identified earlier in this section with the exception of the *stopped* state which is not used. The states are:

NO_USER
IN_AUTH
IN_TP_MENU
USER_RUN

As a user transitions from one state to the next this variable is updated and is used by the *tp_getty* program to determine the appropriate response to user input.

The session structure maintains information about the user's open sessions.

```
struct session_struct {
        session_tty
        session_pid
        session_fd
} sessions
```

The session_structure is used to capture TTY, PID, and the TTY file descriptor of each session. This information is used to suspend, restore and terminate all the processes associated with a particular user session.

G. SUPPORTING MECHANISMS

The keyboard driver was modified to capture the simultaneous pressing of the control and break keys. This action causes the *handle_tp_sak* function to be called in the kernel. The keyboard driver modification is described in Appendix E.

The SE Linux initialization process *init* is modified to start TP_User. No standard *getty* is started. This modification occurs in the *inittab* file.

VII. TRUSTED PATH IMPLEMENTATION

A. INTRODUCTION

The details of the Security Enhanced Linux trusted path implementation are described at a high level in this chapter. Additional implementation details are described in appendices as noted.

B. CONSOLE TERMINOLOGY

The term console is used in the context of computers with different meanings. Zimmer [26] discusses the origins and history of the term console. According to Zimmer there are at least four meanings of console:

- An instrument panel connected to a computer,
- A display/keyboard unit,
- A display/keyboard unit with additional instrumentation used to monitor system status,
- A display mode for a monitor device.

Additionally, the terms TTY and terminal are commonly used to describe a confined display and keyboard unit. Multiple terminals connected to a single computer used to be the common operating mode. In the early days of computers, these terminals were teletype machines which resulted in the abbreviation of TTY for a terminal.

With the advent of modern desktop computing, the virtual console was developed. Here a single computer and the associated display/keyboard unit have the ability to emulate multiple terminals. For example, in Linux it is common to have multiple virtual consoles that are displayed on a single monitor and which can be toggled between by special keystrokes. Each virtual console can have a different user logged in and may perform different functions.

To further confuse the issue, there is the concept of a terminal window. This is a software program that emulates a terminal inside of a graphic user interface such as *X Windows*.

For the sake of this paper the following definitions of terminal related terms will be used.

Console will be reserved for the special case where the primary function of a keyboard/monitor is the administration and/or security of one or more systems.

TTY will be used to refer to the logical virtual console. This choice is driven by the TTY construct in Linux, for example, the command *tty* will return the virtual console that the command was entered on.

Terminal will be reserved for a physically separate keyboard/monitor that is attached to a computer. In the common desktop computer environment there will only be one terminal per system.

The phrase *terminal window* will describe a terminal emulator inside of a graphical user interface. An example of a terminal window is a program like *gnome-terminal* that provides a *TTY*-like environment as a window inside an X Windows session.

The phrase *computer monitor* will be used to describe the physical display device, e.g., computer CRT or flat panel display unit. The phrase was selected primarily to avoid confusion with the phrase *reference monitor* which is discussed in previous chapters.

C. INTERCEPTING OF THE SECURE ATTENTION KEY (SAK)

Securely intercepting the SAK is critical to providing a trusted path. Once the SAK is pressed the trusted path must always be invoked. Untrusted software must not be able to prevent the trusted path from receiving the SAK notification. In order to achieve this in SELinux, the keyboard driver was modified to notify the kernel module of the trusted path when the SAK keystrokes are made.

Linux Magic System Request Key is included in current versions of the Linux kernel and is contained in the sysrq.c file. One of the functions that is provided by the *Magic System Request* is a SAK. This implementation of a SAK is very aggressive and essentially kills all of the processes on the TTY. This was not suitable for our implementation of a trusted path. Also the *Magic System Request* did not work at all within *X Windows* on the systems used for this research. Some amount of time was spent

attempting to determine the cause of the *X Window* problem but it was never precisely identified. Both the aggressive nature of the *Magic System Request* SAK and the *X Windows* issue drove the development of a new SAK

The keyboard generates scan codes that indicate what keys have been pressed or released. This information is passed through various hardware specific drivers to the general keyboard driver. All scan codes eventually arrive at the keyboard driver, so this is a logical point at which to insert the SAK detection functionality. Changing or modifying any of the software drivers between the physical keyboard and the keyboard driver requires recompiling and re-installing the kernel which in turn requires root administrative privileges. Assurance that the SAK cannot be intercepted by malicious code is provided because the kernel would require modification to permit this.

The source code of the function which contains the SAK interception code is provided in Appendix E.

D. KERNEL MODULE OF THE TRUSTED PATH

The trusted path kernel module is primarily a helper set of functionality for the user space trusted path program. This module is a communication channel between the kernel and the trusted path user space program. In addition, it performs some process management.

There are two primary communication systems that are used by this module. For communication from the kernel, signals are used to notify the user space program that a SAK has arrived. Signals are a primitive form of interprocess communication (IPC) and are somewhat limited in their capabilities. More sophisticated methods of IPC were not employed primarily due to the inherent complexity of these methods. This is ultimately becomes a design choice between simplicity and functionality. For this implementation, the simplicity of signals was more important than additional functionality offered by modern IPC methods. The other communication method used in this implementation was a system call for sending messages from the trusted path user space program to the kernel. The trusted path system call is described below.

The kernel module also suspends, restores and terminates processes based on information received by the trusted path system call. The system call provides a process ID (PID) of the group of processes to be suspended, restored or terminated. Using the trusted path ID described below, all of the related processes are identified and the appropriate action is performed by the kernel module.The source code of the trusted path kernel module is provided in Appendix B.

E. CREATION OF A TRUSTED PATH PARENT ID

In this design all user processes are suspended upon entering the trusted path. A reliable method to identify all the processes associated with a user is required to implement this approach. However, Linux allows changing process parentage information in the process table, *task_struct*.

To provide the information needed by the trusted path, an additional field was added to the process table. This field is an integer process ID labeled as *tp_id* for trusted path ID. This field is populated during the *fork* process by the kernel with the *tp_id* of parent process unless the grandparent process ID (PID) is 1 in which case the tp_id is the current process PID. This will result in all processes associated with a user session having the same *tp_id*.

The changes to the original Linux source files required for the trusted path parent ID are detailed in Appendix D, Schedule Modification and F, Fork Modification.

F. ADDITION OF A TRUSTED PATH SYSTEM CALL

The trusted path design consists of two separate groups of code. One group runs in kernel space and the other is a user space construct. To allow the requisite communication to occur, there was a need to build a mechanism for the user space trusted path process to communicate with the kernel. For the sake of simplicity a trusted path system call was implemented. For a discussion on the benefits and disadvantages of using system calls see Love [27].

As identified in the previous chapter, there are four distinct messages that the user space trusted path must be able to send to the kernel:

- Registration of the user space trusted path,
- Suspend user processes,

- Restore user processes, and
- Terminate user processes.

A single system call was designed to allow these four messages be sent to the kernel trusted path mechanism. Detailed implementation information is contained in Appendix C.

G. REPLACEMENT OF THE GETTY AND LOGIN PROCESSES

In the standard SE Linux implementation, the init process will start a small number of *getty* processes. In the SE Linux version used for this research, the getty program is named *mingetty* because it is a minimal version of a getty without some features found in other *getty* programs. *Mingetty* initializes the various TTYs and waits for a user to enter his username. *Mingetty* then *execs* to the *login* program passing the username as a variable. *Login* authenticates the user. *Pluggable Authentication Modules* (PAM) are typically used in modern versions of Linux, including the research version of SE Linux, to perform the authentication.

As discussed in Chapter II, SE Linux uses a security context to limit the behavior of processes. In addition to the standard Linux *user* and *group* identity, the SE Linux security context has an identity, a role and a domain which must be correctly set prior to the user having an interactive shell. If PAM is being used as the authentication method then an additional SE Linux PAM function is used to set the correct security context. After the user is authenticated and appropriate context is set, *login* forks a *bash* shell and the user gains control of the process.

The trusted path implementation replaces *mingetty* and *login* with a trusted path program titled *tp_getty*. This approach to the TTY initialization and user authentication was driven by the state-machine vision of the trusted path described high level design chapter. The vision of the trusted path required the ability of the trusted path to control the state changes which required modification of the process. Another advantage of this process is the reduction in amount of code needed to perform these functions. As an example, in order to support the various authentication methods that are possible, the standard *login* program is in excess of 1400 lines. *Tp_getty* is approximately 500 lines shorter.

The init process starts the number of *tp_getty* processes determined by the system administrator in the init configuration file, *inittab*. The *tp_getty* initializes the *ttys* and waits for the user to press the SAK key combination. Once the SAK is pressed *tp_getty* is notified via a signal sent by the kernel. Then *tp_getty* uses PAM to authenticate the user. Upon successful authentication, the *tp_getty* displays the trusted path menu for the user to select the desired action. For this demonstration project the user may select to quit a session, start a new shell or to restore a shell. The behavior of quitting or restoring a session is dependent on the number of open shells. If there is only one open shell then quitting will log the user out of the system otherwise he will be queried as to which shell should be closed. Restoration of a shell displays similar behavior. The total number of permissible open shells is limited to an arbitrary *MAX_TP_SHELLS*.

Starting a new shell *forks* a new process, the SE Linux context is set for the new process, and a new *tty* is generated. The SE Linux context is set on each shell that is started from the trusted path menu to allow a user to open multiple shells in different contexts. This is accomplished by using the SE Linux PAM *set credentials* function during the *fork* process. After setting the context, the user is switched to the new *tty* and he assumes control of the process. Pressing the SAK key combination while in the user process causes the user's processes to be suspended and the user returns to the *tty* containing the trusted path menu.

There were many challenges to the *tp_getty* implementation. One was the complexity of the Linux *login* code. To support all the various forms of authentication such as Kerberos, crypto cards, PAM and shadow files a significant number of *#ifdef* statements are included in the code. This is an efficient means of updating or modifying an existing piece of code, but these additions make it more complex and difficult to follow.

The most complex challenge was associated with the SE Linux security policy. SE Linux uses various configuration files to generate policy rules that describe the access permissions for subjects and objects. This research used the NSA example SE Linux security policy which contained in excess of 250,000 rules. Determining how to provide *tp_getty* with the correct access permissions was time consuming.

In this implementation of the trusted path, the new *tp_getty* file was given the context of *system_u:object_r:login_exec_t*. This security context allows *tp_getty* to fork processes, use PAM modules, reset TTYs and perform other typical login tasks. Also the *init* domain was modified to allow the *init* process to automatically change it's security domain to *login_exec_t* thus allowing init to start the *tp_getty*. The appropriate configuration files are contained in Appendix H.

H. PROCESS SUSPENSION AND RESTORATION

As part of the design, all user processes must be inactive when the user is interacting with the trusted path menu. This is to prevent the possibility of a user process intercepting any communication between the user and the trusted path.

In order to allow a process to be placed in a suspended state, the process must be removed from the correct run queue which is a data structure that the kernel uses to manage process scheduling. Two additional suspended queues were created in *sched.c* to store the processes that were removed from the run queues. Additionally, the kernel has the ability to temporarily store inactive processes in other queues which required the addition of a *do_not_activate* flag in *sched.h*. This combination allows the trusted path to inactivate the user processes. In general, all user processes are suspended prior to allowing access to the trusted path.

After completion of the user interaction with the trusted path, the user may wish to restore a session. This situation is somewhat more complicated than the suspend process because the user may have multiple sessions that are suspended. The user could have zero, one or more than one open session.

If the user selects restore from the trusted path menu but does not have an open session a brief error statement is displayed and the user returns to the trusted path menu. If the user has just one open session then that session is automatically restored. Otherwise, there are multiple open sessions and the trusted path queries the user to determine which session to open.

I. CONTROLLING THE TTY

In this implementation, the trusted path menu is displayed on TTY1. As a user opens new sessions, the session is displayed on next available TTY. In Linux it is

possible for a user to switch between TTYs with certain keystrokes. To prevent the user from switching between a session TTY and the trusted path menu or vice versa it was necessary to block the ability of the user to switch between TTYs by locking the switch functionality. The trusted path controls the active TTY. Trusting the trusted path

The kenel needs to know that the trusted path system calls are indeed coming from the trusted path menu. Otherwise other software could impersonate the trusted path with the potential of denial of service problems. The approach used in this research is simplistic and there is a potential problem. The trusted path menu registers with the kernel prior to the computer becoming available for user input. The kernel then sets a flag to indicate that the trusted path is registered and to prohibit other processes from registering as the trusted path. Once this flag is set it is not possible to reset the trusted path registration.

The kernel uses this registration to validate that each system call is coming from the trusted path by comparing the PID of the calling process against the previously registered PID of the trusted path.

The flaw in this approach is that if the trusted path menu exits due to some error the init process will restart the trusted path menu program but it will not be able to register or signal the kernel. The only remedy in this situation is to reboot the computer system. This is not a robust solution to the problem and requires additional research.

VIII. TRUSTED PATH TEST PROCEDURES

A. INTRODUCTION

These test procedures are designed to demonstrate that the trusted path is correctly designed and implemented. The testing will demonstrate the correct implementation of the functional supporting requirements as defined earlier. This chapter describes the test methodology for each of the 10 functional sub-requirements (FSR). An action may test more than one FSR, but, for the sake of completeness, the action will be listed under each FSR that is appropriate.

B. FSR 1

1. FSR 1 Purpose

The TSF will provide a trusted path menu to allow the user to select the appropriate action to be performed.

2. FSR 1 Testing Rationale

The trusted path menu should always be displayed in three cases. The menu should display after a user has successfully authenticated. It should display when the SAK is pressed in trusted path state *user_run*. It should also display if an incorrect selection is made from the trusted path menu. The menu should not display at any other time.

3. FSR 1 Test 1

Correctly authenticate to the system in trusted path state *in_auth*. The trusted path menu should display.

4. FSR 1 Test 2

Incorrectly authenticate to the system in trusted path state *in_auth*. The trusted path menu should not display.

5. FSR 1 Test 3

Press the SAK while in the trusted path state *user_run*. The trusted path menu should display.

6. FSR 1 Test 4

Press various key combinations that are not the SAK while in the trusted path state *user_run*. The trusted path menu should not display.

7. FSR 1 Test 5

Press various key combinations that are not the SAK while in the trusted path state *no_user*. The trusted path menu should not display.

8. FSR 1 Test 6

Press the SAK while in the trusted path state *no_user*. The trusted path menu should not display.

9. FSR 1 Test 7

Press various key combinations that are not the SAK while in the trusted path state *in_TP_menu*. The trusted path menu should display.

10. FSR 1 Test 8

Press the SAK while in the trusted path state *in_TP_menu*. The trusted path menu should display.

11. FSR 1 Test 10

Press any single key that is not a trusted path menu selection while in the trusted path state *in_TP_menu*. The trusted path menu should display.

B. FSR 2

1. FSR 2 Purpose

The TSF will suspend all non-trusted path user processes associated with the user upon entering the trusted path.

2. FSR 2 Testing Rationale

In order to assure that no other process can imitate the trusted path or intercept user-TSF communications, all processes associated with the user should be suspended. This suspension must be complete prior to initiating the trusted path menu. Testing of process states will require additional code to log process status and the time the status changes to determine that all of the necessary processes are suspended prior to initiating the trusted path menu. The test code will execute immediately prior to starting the trusted path menu. The trusted path process is considered to be a system process.

3. FSR 2 Test 1

Enter the trusted path menu by successfully authenticating in trusted path state *in_auth*. The logs should indicate that only system processes are running prior to display of the menu.

4. FSR 2 Test 2

Enter the trusted path menu by pressing the SAK in trusted path state *user_run*. The logs should indicate that only system processes are running prior to display of the menu.

5. FSR 2 Test 3

Enter the trusted path menu by selecting any incorrect selection from the trusted path menu in trusted path state *in_TP_menu*. The logs should indicate that only system processes are running prior to display of the menu.

C. FSR 3

1. FSR 3 Purpose

The communication path between the keyboard or computer monitor and the TSF shall be under control of TSF while the trusted path while it is active.

2. FSR 3 Testing Rationale

This functional sub-requirement does not readily support automated testing.

3. FSR 3 Test

Design and code review will be performed by the Thesis advisors. This review should not reveal any potential communication path vulnerabilities.

D. FSR 4

1. FSR 4 Purpose

The TSF will always be notified of the secure attention key (SAK) signal.

2. FSR 4 Testing Rationale

The TSF must always receive the SAK signal. Some of the following tests will also be conducted while evaluating different FSRs and are repeated here for the sake of completeness or because the desired result may be viewed from a different perspective.

3. FSR 4 Test

Press the SAK while in the trusted path state *user_run*. The trusted path menu should display.

4. FSR 4 Test 2

Press various key combinations that are not the SAK while in the trusted path state *user_run*. The trusted path should not be invoked.

5. FSR 4 Test 3

Press the SAK while in the trusted path state *no_user*. The trusted path authentication process should begin.

6. FSR 4 Test 4

Press various key combinations that are not the SAK while in the trusted path state *no_user*. The trusted path should not be invoked.

7. FSR 4 Test 5

Press the SAK while in the trusted path state *in_auth*. The trusted path login process should restart.

8. FSR 4 Test 6

Press various key combinations that are not the SAK while in the trusted path state *in_auth*. The trusted path should not be invoked and depending on the keys pressed the authentication should fail.

9. FSR 4 Test 7

Press the SAK while in the trusted path state *in_TP_menu*. The trusted path menu should continue to be displayed.

10. FSR 4 Test 8

Press various key combinations that are not the SAK while in the trusted path state *in_TP_menu*. The trusted path menu should continue to be displayed.

E. FSR 5

1. FSR 5 Purpose

User authentication will be controlled via the trusted path.

2. FSR 5 Testing Rationale

This functional sub-requirement does not readily support automated testing.

3. FSR 5 Test 1

Design and code review will be performed by the Thesis advisors. This review should indicate that the only entry point to the authentication mechanism is through the trusted path.

F. FSR 6

1. FSR 6 Purpose

Pressing the SAK when there is no authenticated user will result in the TSF initiating an authentication sequence.

2. FSR 6 Testing Rationale

These are the same tests as FSR 4 test 3 and FSR 4 test 4.

3. FSR 6 Test 1

Press the SAK while in the trusted path state *no_user*. The trusted path authentication process should begin.

4. FSR 6 Test 2

Press various key combinations that are not the SAK while in the trusted path state *no_user*. The trusted path should not be invoked.

G. FSR 7

1. FSR 7 Purpose

Failed authentication will result in returning to the initial *no_user* state.

2. FSR 7 Testing Rationale

Failed authentication will cause the computer to return to the *no_user* state and will require the user to begin the process over.

3. FSR 7 Test 1

Incorrectly authenticate to the system in trusted path state *in_auth*. The message to press the SAK to begin authentication should be displayed.

4. FSR 7 Test 2

Correctly authenticate to the system in trusted path state *in_auth*. The trusted path menu should display.

H. FSR 8

1. FSR 8 Purpose

Successful authentication will result in the trusted path menu being displayed.

2. FSR 8 Testing Rationale

These are the same tests as in FSR 7.

3. FSR 8 Test 1

Incorrectly authenticate to the system in trusted path state *in_auth*. The message to press the SAK to begin authentication should be displayed.

4. FSR 8 Test 2

Correctly authenticate to the system in trusted path state *in_auth*. The trusted path menu should display.

I. FSR 9

1. FSR 9 Purpose

Selecting *run* from the trusted path menu will restore any previously suspended user processes or cause a new shell process to be spawned.

2. FSR 9 Testing Rationale

The trusted path must restore any previously suspended processes upon return to the trusted path *user_run* state.

3. FSR 9 Test 1

While in the trusted path *user_run* state initiate some number of processes. Perform a process listing and save the output to a file. Press the SAK and when in the trusted path menu select run to return to the *user_run* state. Again perform the process listing and save the output to a different file. Compare the two files and the only difference should be the process listing command which might have a different process ID.

J. FSR 10

1. FSR 10 Purpose

Selecting *exit* from the trusted path menu will terminate any user processes and cause the state to change to *no_user*. The TTY will display the pre-authentication message requesting potential users to press the SAK.

2. FSR 10 Testing Rationale

This will require additional code to generate test results after the user has logged out. This code will log running processes after completing the user logout.

3. FSR 10 Test 1

From the trusted path menu select exit then log back into the system and proceed to the trusted path *user_run* state. Review the logs to ensure that all user processes were stopped when the *no_user* state was reached.

THIS PAGE INTENTIONALLY LEFT BLANK

IX. CONCLUSIONS

A. ANALYSIS AND DISCUSSION

1. The Increasing Complexity of Software

As discussed previously in this paper, the complexity of software increases the likelihood of security relevant issues in the design and implementation. Complexity also decreases the chance that an individual will be able to discover a particular flaw. With enough complexity it may not be possible to determine whether a specific piece of code has a security issue in the larger context of the code it interacts with.

Most software in common usage today is being developed in an incremental fashion. An example of this is the Microsoft Windows XP operating system. The core operating system is modified from Windows 2000 which was modified from Windows NT 4.0. This approach to development is efficient and follows the tenant of code reuse that is promoted by the software development community but typically results in much more complex code. This occurs because additional functionality is inserted into the previous code. This additional code adds features but can significantly increase the complexity of a group of code and in fact can make it much more difficult to understand what are the direct results and side-effects of any given code.

It can be argued that this evolutionary process is the only economical approach to develop large software packages. While this is reasonable, little formal thought is given to the security implications. As the code becomes more complex, the likelihood of significant security problems increases. There is a real economic impact of security issues both in the traditional computer security sense and the more mundane sense of having to pay system administrators to constantly patch and maintain systems.

Linux is a very complex set of software packages. Most of the core kernel functions have been incrementally modified by many authors to offer an increasing number of functions. As an example of this, the scheduling code of the 2.6 kernel is 2,916 lines in contrast to the 849 lines that were in the 1.0.9 kernel. Any particular thread of execution through the scheduler is understandable but the cumulative effects of all the

code makes it difficult to understand the scheduler behavior in a meaningful and holistic sense. This is just one example but the remainder of the kernel contains similar complexity issues. Adding applications software on top of the operating system explodes the difficulty of understanding the security relevant software interactions.

Security Enhanced Linux (SELinux) is a security overlay on the traditional Linux. Security-relevant enforcement mechanisms have been added with one goal being flexibility. The flexibility of the security mechanism significantly increases the complexity of the software. This complexity is in addition to the complexity of the underlying operating system. There is a likelihood approaching certainty that significant security problems exist in this system, though this is not different from other common operating systems.

The Common Criteria [29] mandates the reduction of complexity as a key factor in developing assurance:

> Design complexity minimisation contributes to the assurance that the code is understood — the less complex the code in the TSF, the greater the likelihood that the design of the TSF is comprehensible. Design complexity minimisation is a key characteristic of a reference validation mechanism.

The primary purpose of minimizing the complexity of a system is to allow understanding of how the computer system enforces security relevant policy. To state that a computer is high assurance demands that the security relevant functions of the system are understood through the range of potential inputs and the functions will always behave correctly in terms of the security policy. This is the fundamental idea behind the reference monitor concept proposed by Anderson [2]. Related to this complexity issue is the intermingling of security functions with the remainder of the kernel code. With security functions scattered about in thousands of lines of other code, it becomes very difficult to understand the behavior of these functions.

2. Security Enhanced Linux

SELinux is an interesting research project. The addition of mandatory access controls may provide some additional security to the system depending on the SELinux policy used.

The SELinux example policy that is provided by NSA is a very liberal policy that seems to be designed to minimize potential problems for the user. By default, SELinux will deny an action unless that action is specifically allowed in the policy configuration. This is a correct approach to a secure system but in the provided example policy most common actions are specifically allowed. In the example policy provided with this version of SELinux there were in excess of 250,000 rules that permitted actions. In contrast, a typical firewall or VPN implementation may have 100 rules. This implies that a great amount of work would be required to modify this rule set correctly for a given situation.

As long as SELinux is viewed as a research project it has value. It is not high assurance and would have difficulty in being classified even as medium assurance under the Common Criteria. It should not be deployed as a solution in situations that require considerable assurance.

3. Adding Security to Existing Systems

Most of the effort of this project was in understanding the existing implementation of SELinx and the underlying Linux system. Due to the complexity of the current implementation and the lack of design documentation it was difficult to determine how and where to insert pieces of functionality. It was also impossible to determine the potential effects from other parts of the existing code on the new trusted path. This results in some doubt as to the security of the trusted path implementation specifically in terms of potential issues arising from within the kernel. Most of code that was identified as relevant was examined but that only represents a small fraction of the code base and issues such as timing problems were not practical to evaluate. At the core of the issue is that this implementation depends on a tremendous amount of kernel code. The trusted path requires that the kernel behave correctly. This places a great deal of trust in unknown code and programmers.

It is axiomatic that adding security to an existing system is much more expensive than incorporating security from the beginning. This demonstration project demonstrates the truth of this statement. The effectiveness of adding security to existing software decreases as the complexity increases. At some point the complexity becomes

unmanageable and regardless of the efforts to add security very little is accomplished because the existing code base is too large and not understandable with respect to the added security functionality. This statement is empirically demonstrated by the software security patch that results in additional security problems.

These observations imply that any common operating systems in use today will have significant security problems. The code bases of Windows, Linux and Unix are too large and complex to prevent potential security issues. The original implementation of these products did not have security as a primary design consideration. If the lives of DoD personnel depend on the security of data contained in a computer then a high assurance system is needed and none of the listed operating systems will meet the requirements.

B. FUTURE WORK

Ensuring that all trusted path system calls are originating from the trusted path menu requires additional research. The current implementation provides rudimentary security from processes impersonating the trusted path but may require rebooting of the computer if the trusted path menu experiences an error. Additionally, the design of the trusted path identifier, *tp_id,* is very dependent upon the *task* structure. Further research might provide a more robust implementation mechanism for this identifier.

As currently implemented, the trusted path has been subjected to minimal code review and testing. As a single individual with little oversight it is very possible that this work contains some errors. An extensive code review and more rigorous testing of the implementation must be conducted prior to use of this software in a production system.

APPENDIX A. TRUSTED PATH –USER SPACE

This is the source code for the trusted path user space module. It is derived from a number of sources. This represents the majority of the coding effort for this demonstration project.

```
//  Trusted path user space program
//
//  This program acts as the trusted path
//  for selinux
//
//  This implementation is part of a
//  masters thesis
//
//  ahilchie
//
//  There are major sections which are derived
//  from the standard login.c and mingetty.c used by RedHat
//

#include <sys/param.h>
#include <stdio.h>
#include <ctype.h>
#include <unistd.h>
#include <getopt.h>
#include <memory.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/file.h>
#include <termios.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/wait.h>
#include <signal.h>
#include <errno.h>
#include <grp.h>
#include <pwd.h>
#include <utmp.h>
#include <setjmp.h>
#include <stdlib.h>
#include <string.h>
#include <sys/syslog.h>
```

```
#include <sys/sysmacros.h>
#include <netdb.h>
#include "pathnames.h"
#include "my_crypt.h"
#include "login.h"
#include "xstrncpy.h"
#include "nls.h"

// additional includes from mingetty
#include <fcntl.h>
#include <stdarg.h>
#include <syslog.h>
#include <sys/utsname.h>
#include <time.h>
#include "userspace.h"

// additional includes for vt
#include <dirent.h>
#include <sys/vt.h>
#include <sys/types.h>

// selinux relevant
#include <selinux/selinux.h>
#include <sys/sysmacros.h>
#include <linux/major.h>
#include <utmp.h>
#include <security/pam_appl.h>
#include <security/pam_misc.h>

// PAM
#  define PAM_MAX_LOGIN_TRIES         3
#  define PAM_FAIL_CHECK if (retcode != PAM_SUCCESS) { \
      fprintf(stderr,"\n%s\n",pam_strerror(pamh, retcode)); \
      syslog(LOG_ERR,"%s",pam_strerror(pamh, retcode)); \
      pam_end(pamh, retcode); exit(1); \
   }
#  define PAM_END { \
          pam_setcred(pamh, PAM_DELETE_CRED); \
          retcode = pam_close_session(pamh,0); \
          pam_end(pamh,retcode); \
}

#include <lastlog.h>
#include "setproctitle.h"
```

```
#ifdef USE_TTY_GROUP
#  define TTY_MODE 0620
#else
#  define TTY_MODE 0600
#endif

#ifndef MAXPATHLEN
#  define MAXPATHLEN 1024
#endif

#define VTNAME "/dev/tty%d"
#define TP_VT 1

// State names for the trusted path
#define NO_USER 0  // no authenticated user on system
#define IN_AUTH 1   // user in authentication process
#define IN_TP_MENU 2       // user authenticated and tp in control
#define USER_RUN 3         // user authenticated and tp not in effect

// trusted path state
int tp_state = NO_USER; // by default we start w/o a user

struct  passwd *pwd, pwdcopy;

#define MAX_TP_SHELLS 10 // arbitrary max number allow shells
// A simple session struct
struct session_struct {
        int session_tty;
        int session_pid;
        int session_fd; //the tty fd
} sessions[MAX_TP_SHELLS];

int num_open_sessions = 0; // start with no open sessions
int current_session = 0;

// a couple of functions for dealing with the open session array
void add_session(int new_tty, int new_pid, int new_fd);
void delete_session(int session_num);
void print_array(void);

char ** env;
static char *username;
static char *user_dir;
char    *hostname;
```

```
static char *progname;
static char *mg_tty; //mingetty tty
static pid_t pid;
static int noclear = 0;
static int nohangup = 0;
extern char **environ;

pam_handle_t *pamh = NULL;

int retcode;      //return code for pam function

// standard pam conversion function which
// provides a means to pass text to user from pam
struct pam_conv conv = { misc_conv, NULL };

// error loggin function
static void error (const char *fmt, ...);

// mingetty standard tty initialization routine
static void open_tty (void);

// trusted path signal handling routine
void tp_signal_catcher(int the_sig);

// trusted path routine to interact with pam
// returns 0 on successful completion
// any nonzero return indicates failure of the
// the function or pam
int tp_auth(void);

// displays a message to the user to press the SAK
void display_SAK_msg(void);

// The trusted path menu charater catching function
// valid return values are:
// s to start a new session
// q to terminate a session
// r to restore a session
char do_tp_menu(void);

// printf a bunch of endlines to clear the screen
void clear_screen(void);

// printf of the menu choices
void print_tp_menu(void);
```

```
// error statement if user failed to select
// a valid menu choice
void print_tp_menu_error(void);

// trusted path function to
// switch to and open a new tty
// start a new shell
// returns zero on successful completion
int tp_open_shell(void);

// trusted path function to switch the
// active tty to int vtno
// returns a zero on successful completion
int switch_vt(int vtno);

// trusted path function to
// allow the user to pick a session to
// terminate or restore
// returns the session number
int  get_session_num(void);

// trusted path function to
// terminate a session
// returns a zero on successfull completion
int tp_terminate_session(void);

int main(int argc, char **argv){

        // set the program name for error logging
        progname = argv[0];
        if (!progname)
                progname = "trustedpath";

/*      //remove this to set up the initial
        //vt locking to prevent users from switching
        //between ttys prior to initial login

      int tmp_fd;
        tmp_fd = open("/dev/console",O_WRONLY,0);
      ioctl(tmp_fd, VT_LOCKSWITCH, 256);
*/


        // set up the signal structures so we can
```

```
        // catch kernel sak msg
    struct sigaction tp_action;
    tp_action.sa_handler = tp_signal_catcher;
    tp_action.sa_flags = 0;

    if (sigaction(SIGUSR1, &tp_action, NULL) == -1) {
        perror("SIGNIT");
        return 1;
    }

        pid = getpid ();
        putenv ("TERM=linux");

        //tty argv is passed in from inittab
        // as is in the form of tty1
        mg_tty = argv[optind];

        if (!mg_tty)
                printf("no tty");

        if (strncmp (mg_tty, "/dev/", 5) == 0)
                mg_tty += 5;

        trustedpath(REGISTER, pid);

        open_tty ();

// the primary trusted path decision loop
for ( ; ;) {

  int my_ret = 0;
  char menu_ret = NULL;

  switch (tp_state) {

    case NO_USER:
        display_SAK_msg();
                pause();                // wait for sak
                tp_state = IN_AUTH;         // change state
                break;                  // break to next state

    case IN_AUTH:
        my_ret = tp_auth();    //authenticate the user

                if(my_ret)              // authentication failed
```

```
                tp_state = NO_USER;
        else
                tp_state = IN_TP_MENU; //change state

        break;                  // break to next state

case IN_TP_MENU:
        menu_ret = do_tp_menu();

        //start a new shell
        if (menu_ret =='s') {

                // check the current shell count
                if(num_open_sessions >=MAX_TP_SHELLS){
                        printf("There are too many open shells\n");
                        printf("Please close a shell to proceed\n");
                        sleep(3);
                }
                else{
                        tp_state = USER_RUN;
                        tp_open_shell();
                }
                break;
    }

        //quit a session
        else if(menu_ret == 'q') {

                // find out which session to end
                int my_session = 0;
                my_session = get_session_num();

                // if there are no open sessions
                if (!my_session){
                        printf("There is no open session\n");
                        sleep(5);
                        // don't change state eg stay in tp_menu
                }

                else {

                        tp_terminate_session(my_session);
                        // check if that was the last session
                        if(! num_open_sessions){
                                tp_state = NO_USER;
```

```
                                        PAM_END;
                                }
                        }
                        break;
                }

                //resume a previous shell
                else if (menu_ret =='r'){
                        current_session = get_session_num();
                        if(!current_session){   // no open session
                                printf("There is no open session to restore\n");
                                break;
                        }

                        // restore the suspended processes
                        trustedpath(RESTORE,            sessions[current_session-
1].session_pid);
                        switch_vt(sessions[current_session-1].session_tty);
                        tp_state = USER_RUN;
                }

                break;

    case USER_RUN:
                //wait for a sak
                pause();

                // after pause reactive tp menu
                // suspend user processes
                trustedpath(SUSPEND, sessions[current_session-1].session_pid);

                current_session = 0;

                // switch to trusted path tty
                switch_vt(TP_VT);

                // reset state
                tp_state = IN_TP_MENU;
                break;


   } //end switch
  } //end of endless loop -- can i say that?
}
```

```
// Terminates a single user session
int tp_terminate_session(my_session){

        // trusted path kill sys call
        trustedpath(KILL, (int) sessions[my_session-1].session_pid);
        sleep(1);
        // clean up the data structs
        delete_session(my_session);

        // need to close tty
        ioctl(sessions[my_session].session_fd,
                VT_DISALLOCATE, sessions[my_session-1].session_tty);
        ioctl(sessions[my_session].session_fd,
                VT_RELDISP, sessions[my_session-1].session_tty);

        close(sessions[my_session-1].session_fd);


        return 0;
}

// tp_open_shell opens a virtual terminal
// and then creates a shell for the user
// this code is derived from Jon Tomb's open

int tp_open_shell(void){

        // set up the new tty
        struct vt_stat vt;
        int fd = 0;
        int new_pid;
        int vtno = -1;
        char vtname[sizeof VTNAME + 2];

        // we don't have a tty selected
        if (vtno == -1) {

                // get a 'console' fd to query vt database
                if ((fd = open("/dev/console",O_WRONLY,0)) < 0) {
                perror("Failed to open /dev/console\n");
                return(2);
                }

                // grab the next available vt
                if ((ioctl(fd, VT_OPENQRY, &vtno) < 0) || (vtno == -1)) {
```

```
        perror("Cannot find a free VT\n");
        close(fd);
        return(3);
        }

        // get a handle to the vt state struct
        if (ioctl(fd, VT_GETSTATE, &vt) < 0) {
        perror("can't get VTstate\n");
        close(fd);
        return(4);
        }

        // unlock the vt switching functionality

        if (ioctl(fd, VT_UNLOCKSWITCH, 256) < 0) {
                perror("Unable to unlock VT\n");
                close(fd);
                return(5);
        }
}

sprintf(vtname, VTNAME, vtno);

char my_tty[sizeof VTNAME +2];
sprintf(my_tty, "tty%d", vtno);

// fork a new process to use for the
// user terminal
if((new_pid=fork()) == 0) {
        /* leave current vt */
        signal(SIGINT, SIG_DFL);

        // finish pam initialization
        // This will call the selinux pam module
        // which will set the context for the session
        retcode = pam_set_item(pamh, PAM_RHOST, hostname);
        retcode = pam_set_item(pamh, PAM_TTY, my_tty);

        retcode = pam_open_session(pamh, 0);
        PAM_FAIL_CHECK;

        retcode = pam_setcred(pamh, PAM_ESTABLISH_CRED);
        PAM_FAIL_CHECK;

        //close the password database
```

```
        endpwent();

        //set the group id with the user's gid
        setgid(pwd->pw_gid);

        // if the user doesn't have a defined shell
        if(*pwd->pw_shell == '\0') pwd->pw_shell = _PATH_BSHELL;

        environ = (char**)malloc(sizeof(char*));
        memset(environ, 0, sizeof(char*));

        // set up the environment
        setenv("HOME", pwd->pw_dir, 0);

        if(pwd->pw_uid)
                setenv("PATH", _PATH_DEFPATH, 1);
        else
                setenv("PATH", _PATH_DEFPATH_ROOT, 1);

        setenv("SHELL", pwd->pw_shell, 1);
        setenv("LOGNAME", pwd->pw_name,1);

int i;
env = pam_getenvlist(pamh);

if (env != NULL) {
        for (i=0; env[i]; i++) {
                putenv(env[i]);
       }
}

        if (setsid() < 0) {
        fprintf(stderr, "open: Unable to set new session\n");
        }

        close(0);
        close(1);
        close(2);
        close(fd);
        // open the new vt
        // if this fails we can't tell anyone
        if ((fd = open(vtname, O_RDWR)) == -1) {
        _exit (4); // silently die
        }
```

```
                dup(fd); dup(fd);


                (void) ioctl(fd, VT_ACTIVATE, vtno);
                (void) ioctl(fd, VT_WAITACTIVE, vtno);

                setuid(pwd->pw_uid);

                if (chdir(pwd->pw_dir) < 0) {
                printf(_("No directory %s!\n"), pwd->pw_dir);
                if (chdir("/"))
                                exit(0);
                user_dir = "/";
                printf(_("Logging in with home = \"/\".\n"));
                }

        execlp("/bin/bash", "bash", NULL);

        }

        if ( new_pid < 0 ) {
                PAM_END;
        perror("open: fork() error");
        return(6);
        }

        /* parent */
        signal(SIGHUP, SIG_IGN);
        signal(SIGINT, SIG_IGN);
        signal(SIGQUIT, SIG_IGN);
        signal(SIGTSTP, SIG_IGN);
        signal(SIGTTIN, SIG_IGN);
        signal(SIGTTOU, SIG_IGN);

        // update our database
        add_session(vtno, new_pid, fd);
        // the num_open_sessions is incremented in add_session
        current_session = num_open_sessions;

        // wait a couple of sec then lock the tty
        sleep(1);
        ioctl(fd, VT_LOCKSWITCH, 256);

        return 0;
}
```

```
// Switch between virtual terminals
// return zero if successful
int switch_vt(vtno){

  int fd;

  if ((fd = open("/dev/console",O_WRONLY,0)) < 0) {
      perror("Can't open /dev/console\n");
      return(3);
  }

  // unlock the vt switching functionality
  ioctl(fd, VT_UNLOCKSWITCH, 256);


  if (ioctl(fd, VT_ACTIVATE, vtno) < 0) {
      fprintf(stderr, "Failed to select VT %d (%s)\n", vtno,
            strerror(errno));
      return(3);
  }

  // make sure we have switched
  (void) ioctl(fd, VT_WAITACTIVE, vtno);
  // lock the vt switching functionality
  ioctl(fd, VT_LOCKSWITCH, 256);

  return(0);
}

// display the trusted path menu
// returns the selected value
char do_tp_menu(){

      char letter;
      int error = 0;
      clear_screen();
      print_tp_menu();

      do {
            letter = getchar();
            error ++;
            if( error > 3) {
                  clear_screen();
```

```
                print_tp_menu_error();
                print_tp_menu();
                error = 0;
            }

        } while ((letter != 's') && (letter != 'q') && (letter != 'r'));

            return letter;
}

// a simplistic way to move everything
// off the screen
void clear_screen() {
        int i =0 ;
        for (i = 0; i < 50; i++)
                printf("\n");
}

void print_tp_menu(){
        printf("Type s then hit enter to start a new shell\n");
        printf("Type q then hit enter to quit a session\n");
        printf("Type r then hit enter to resume a session\n");
}

void print_tp_menu_error(){
        printf("you must select the letter s or q or r\n");
}


void display_SAK_msg(void){
            printf("\nPlease press the control and break key at the same time to
begin\n");
}

//          This section is derived from login.c
//
//          2004-7-24 ahilchie@nps.navy.mi
//          using pam to authenticate the user

int tp_auth() {

            char    *tty_name;
            extern int optind;        //used to grab tty from arg

            username = hostname = tty_name = NULL;
```

```
        // set local environment, msg cat dir & gettext domain
        setlocale(LC_ALL,"");
        bindtextdomain(PACKAGE, LOCALEDIR);
        textdomain(PACKAGE);

        setpgrp();

        // start the pam auth procedure
        retcode = pam_start("login",username, &conv, &pamh);

        // bail out if we weren't able to talk with pam
        if(retcode != PAM_SUCCESS) {
                fprintf(stderr, _("login: PAM Failure, aborting: %s\n"),
                        pam_strerror(pamh, retcode));
                syslog(LOG_ERR, _("Couldn't initialize PAM: %s"),
                        pam_strerror(pamh, retcode));
                exit(99);
        }

        // hostname is set to NULL
        retcode = pam_set_item(pamh, PAM_RHOST, hostname);
        PAM_FAIL_CHECK;

        // for root to be able to login the tty name
        // must be set to one of the devices
        // listed in /etc/securetty
        // mg_tty is the tty value in /etc/inittab
        retcode = pam_set_item(pamh, PAM_TTY, mg_tty);
        PAM_FAIL_CHECK;

        // localize the user prompt
        retcode= pam_set_item(pamh, PAM_USER_PROMPT, _("Trusted Path
login: "));
        PAM_FAIL_CHECK;


      int failcount=0;

      // set user name to null
        pam_set_item(pamh, PAM_USER, NULL);

        // the primary authentication loop
      retcode = pam_authenticate(pamh, 0);
      while((failcount++ < PAM_MAX_LOGIN_TRIES) &&
```

```
          ((retcode == PAM_AUTH_ERR) ||
           (retcode == PAM_USER_UNKNOWN) ||
           (retcode == PAM_CRED_INSUFFICIENT) ||
           (retcode == PAM_AUTHINFO_UNAVAIL))) {
        pam_get_item(pamh, PAM_USER, (const void **) &username);

        syslog(LOG_NOTICE,_("FAILED LOGIN %d FROM %s FOR %s,
%s"),
              failcount, hostname, username, pam_strerror(pamh, retcode));

        fprintf(stderr,_("Login incorrect\n\n"));
        pam_set_item(pamh,PAM_USER,NULL);
        retcode = pam_authenticate(pamh, 0);
      }

        // if pam failed capture the reason
      if (retcode != PAM_SUCCESS) {
        pam_get_item(pamh, PAM_USER, (const void **) &username);

        if (retcode == PAM_MAXTRIES)
          syslog(LOG_NOTICE,_("TOO MANY LOGIN TRIES (%d) FROM
%s FOR "
                "%s, %s"), failcount, hostname, username,
                 pam_strerror(pamh, retcode));
        else
          syslog(LOG_NOTICE,_("FAILED LOGIN SESSION FROM %s FOR
%s, %s"),
                hostname, username, pam_strerror(pamh, retcode));

        fprintf(stderr,_("\nLogin incorrect\n"));
        pam_end(pamh, retcode);
        exit(0);
      }

      retcode = pam_acct_mgmt(pamh, 0);

        // prompt the user to change password if it is time
      if(retcode == PAM_NEW_AUTHTOK_REQD) {
        retcode                    =                    pam_chauthtok(pamh,
PAM_CHANGE_EXPIRED_AUTHTOK);
      }

      PAM_FAIL_CHECK;

        // update the username
```

```
        retcode = pam_get_item(pamh, PAM_USER, (const void **)
&username);
        PAM_FAIL_CHECK;

        // if something is wrong with user name -- log it
        if (!username || !*username) {
                fprintf(stderr, _("\nSession setup problem, abort.\n"));
                syslog(LOG_ERR, _("NULL user name in %s:%d. Abort."),
              __FUNCTION__, __LINE__);
        pam_end(pamh, PAM_SYSTEM_ERR);
                exit(1);
        }
        if (!(pwd = getpwnam(username))) {
                fprintf(stderr, _("\nSession setup problem, abort.\n"));
                syslog(LOG_ERR, _("Invalid user name \"%s\" in %s:%d.
Abort."),
             username, __FUNCTION__, __LINE__);
                pam_end(pamh, PAM_SYSTEM_ERR);
                exit(1);
        }

   /*
    * Create a copy of the pwd struct - otherwise it may get
    * clobbered by PAM
    */
   memcpy(&pwdcopy, pwd, sizeof(*pwd));
   pwd = &pwdcopy;
   pwd->pw_name   = strdup(pwd->pw_name);
   pwd->pw_passwd = strdup(pwd->pw_passwd);
   pwd->pw_gecos  = strdup(pwd->pw_gecos);
   pwd->pw_dir    = strdup(pwd->pw_dir);
   pwd->pw_shell  = strdup(pwd->pw_shell);
   if (!pwd->pw_name || !pwd->pw_passwd || !pwd->pw_gecos ||
      !pwd->pw_dir || !pwd->pw_shell) {
         fprintf(stderr, _("login: Out of memory\n"));
         syslog(LOG_ERR, "Out of memory");
         pam_end(pamh, PAM_SYSTEM_ERR);
         exit(1);
   }
   username = pwd->pw_name;

    // Initialize the supplementary group list.
    // This should be done before pam_setcred because
    // the PAM modules might add groups during pam_setcred.
   if (initgroups(username, pwd->pw_gid) < 0) {
```

```
        syslog(LOG_ERR, "initgroups: %m");
        fprintf(stderr, _("\nSession setup problem, abort.\n"));
        pam_end(pamh, PAM_SYSTEM_ERR);
        exit(1);
    }
    return 0;
}

// a function to allow the user to select a session
// if there is only one open session it returns w/o user input
// return is the session number
int  get_session_num(){
        int rc = 0;
        int temp =0;
        clear_screen();
        //printf("in get session num and num open shells is: %d\n",
        //      num_open_sessions);

        switch (num_open_sessions){

                case 0:
                        printf("There is not an open shell\n");
                        printf("Please select s to start a new shell\n");
                        sleep(3);
                        return rc;
                        break;

                case 1:
                        return 1;
                        break;
                default:
                        while(!rc){
                                printf("There are %d open sessions\n",
                                        num_open_sessions);
                                printf("Enter a session number to select it\n");
                                char my_input;
                                my_input = getchar();
                                temp = atoi(&my_input);
                                if( temp > 0 && temp <= num_open_sessions)
                                        rc = temp;
                                printf("That selections is not valid\n");
                        }
                        return rc;
        }
}
```

```
// error() - output error messages
// direct copy from mingetty
static void error (const char *fmt, ...)
{
        va_list va_alist;

        va_start (va_alist, fmt);
        openlog (progname, LOG_PID, LOG_AUTH);
        vsyslog (LOG_ERR, fmt, va_alist);
        /* no need, we exit anyway: closelog (); */
        va_end (va_alist);
        sleep (5);
        exit (EXIT_FAILURE);
}

// open_tty - set up tty as standard { input, output, error }
// direct copy from mingetty
static void open_tty (void)
{
        struct sigaction sa, sa_old;
        char buf[40];
        int fd;

        /* Set up new standard input. */
        if (mg_tty[0] == '/')
                strcpy (buf, mg_tty);
        else {
                strcpy (buf, "/dev/");
                strcat (buf, mg_tty);
        }
        /* There is always a race between this reset and the call to
           vhangup() that s.o. can use to get access to your tty. */
        if (chown (buf, 0, 0) || chmod (buf, 0600))
                if (errno != EROFS)
                        error ("%s: %s", mg_tty, strerror (errno));

        sa.sa_handler = SIG_IGN;
        sa.sa_flags = 0;
        sigemptyset (&sa.sa_mask);
        sigaction (SIGHUP, &sa, &sa_old);

        /* vhangup() will replace all open file descriptors in the kernel
           that point to our controlling tty by a dummy that will deny
           further reading/writing to our device. It will also reset the
```

```
          tty to sane defaults, so we don't have to modify the tty device
          for sane settings. We also get a SIGHUP/SIGCONT.
         */
        if ((fd = open (buf, O_RDWR, 0)) < 0)
                error ("%s: cannot open tty: %s", mg_tty, strerror (errno));
        if (ioctl (fd, TIOCSCTTY, (void *) 1) == -1)
                error ("%s: no controlling tty: %s", mg_tty, strerror (errno));
        if (!isatty (fd))
                error ("%s: not a tty", mg_tty);

        if (nohangup == 0) {
                if (vhangup ())
                        error ("%s: vhangup() failed", mg_tty);
                /* Get rid of the present stdout/stderr. */
                close (2);
                close (1);
                close (0);
                close (fd);
                if ((fd = open (buf, O_RDWR, 0)) != 0)
                        error ("%s: cannot open tty: %s", mg_tty,
                                strerror (errno));
                if (ioctl (fd, TIOCSCTTY, (void *) 1) == -1)
                        error ("%s: no controlling tty: %s", mg_tty,
                                strerror (errno));
        }
        /* Set up stdin/stdout/stderr. */
        if (dup2 (fd, 0) != 0 || dup2 (fd, 1) != 1 || dup2 (fd, 2) != 2)
                error ("%s: dup2(): %s", mg_tty, strerror (errno));
        if (fd > 2)
                close (fd);
        /* Write a reset string to the terminal. This is very linux-specific
          and should be checked for other systems. */
        if (noclear == 0)
                write (0, "\033c", 2);

        sigaction (SIGHUP, &sa_old, NULL);
}

// handle the user 1 signal that the kernel
// sends to notify the trusted path of the SAK
void tp_signal_catcher(int the_sig){
        if (the_sig == SIGUSR1){
                return;
        }
}
```

```
// adds a user session to the sessions array
void add_session(int new_tty, int new_pid, int new_fd){
	sessions[num_open_sessions].session_tty = new_tty;
	sessions[num_open_sessions].session_pid = new_pid;
	sessions[num_open_sessions].session_fd = new_fd;
	num_open_sessions ++;
}

// remove a user session from the sessions array
void delete_session(int session_num){
	//validate the session_num
	if (session_num > num_open_sessions){
		printf("the session number is too large\n");
		return;
	}
	if (session_num < 0){
		printf("the session number is too small\n");
		return;
	}
	//careful the array is zero indexed
	int array_index = session_num - 1;
	//simple case first
	if (array_index == (num_open_sessions - 1)){
		// sort of redundent but ...
		sessions[session_num].session_tty = 0;
		sessions[session_num].session_pid = 0;
		sessions[session_num].session_fd = 0;
		num_open_sessions --;
	}
	else {
		int loopcounter = array_index;
		loopcounter ++;
		while (loopcounter <= num_open_sessions){
			//move the info up a slot
			sessions[loopcounter-1].session_tty =
				sessions[loopcounter].session_tty;
			sessions[loopcounter-1].session_pid =
				sessions[loopcounter].session_pid;
			sessions[loopcounter-1].session_fd =
				sessions[loopcounter].session_fd;
			loopcounter++;
		}
		num_open_sessions --;
	}
}
```

THIS PAGE INTENTIONALLLY LEFT BLANK

APPENDIX B. TRUSTED PATH – KERNEL

These two files, trustedpath.h and trustedpath.c are the kernel portions of the trusted path. Additional kernel modifications were necessary for the project and are documented in the other appendices.

trustedpath.h

```
#ifndef __LINUX_TRUSTEDPATH_H
#define __LINUX_TRUSTEDPATH_H

/*      include/linux/trustedpath.h
 *
 *      ver 1  ahilchie 6-14-04
 *
 *      Demo implementation of Trusted Path for SELinux
 *
 *      based on the magic system request key
 */

#include <linux/config.h>
#include <linux/linkage.h>
#include <linux/sched.h>

#define REGISTER 0
#define SUSPEND 1
#define RESTORE 2
#define KILL 3

struct pt_regs;
struct tty_struct;

/* Trusted Path interface
*/

void handle_trustedpath( struct pt_regs *, struct tty_struct *);
void send_sak_sig(void);
int suspend_all_processes( int user_pid);
int restore_all_processes(int pid);
#endif
```

trustedpath.c

```
//      drivers/char/trustedpath.c
//      ver 1 ahilchie 6-14-04
//      Demo implementation of Trusted Path for SELinux

#include <linux/config.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/tty.h>
#include <linux/mount.h>
#include <linux/kdev_t.h>
#include <linux/major.h>
#include <linux/trustedpath.h>
#include <linux/kbd_kern.h>
#include <linux/quotaops.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/suspend.h>
#include <linux/writeback.h>
#include <linux/buffer_head.h>
#include <asm/ptrace.h>
#include <linux/list.h>
//#include <sys/syscall.h>
//#include <asm/uaccess.h>


unsigned long last_tp_time = 0;
unsigned long tp_delay = 1;

//extern void *sys_call_table[];
//int uid;

//asmlinkage int(*getuid_call)();

//  trusted path userspace module pid
//  this is the only process we will accept
//  system calls from
//  by default we trust init's parent
int tpum = 0;

// lock will be set to one after the first
// registration of the user space trusted path
```

```
// this will prevent new processes trying to
// pretend to be the trusted path
int lock = 0;

// Notify the trusted path in user space that a SAK has arrived
void send_sak_sig(){
        // test to make sure tpum is registered
        if (tpum ==0){
                printk(KERN_CRIT "Received SAK prior to Trusted Path registering");
        }
        else {
                struct task_struct *p;
                p = find_task_by_pid(tpum);
                force_sig(SIGUSR1, p);
        }
}

// Handle the SAK notification from keyboard.c
// The trusted path state is maintained in userspace
// so all the kernel does is signal the trusted path
void handle_trustedpath( struct pt_regs *pt_regs, struct tty_struct *tty){

         // hack to stop multiple responses to single keystroke
        if (get_seconds() <= last_tp_time + tp_delay)
                return;

        last_tp_time = get_seconds();

        send_sak_sig();
}

// Another helper funtion to suspend processes associated
// with a user pid
// It is a little clunky because we are receiving an int
// not a task_struct from user space.  Therefore we end up cycling
// through all the processes
int suspend_all_processes( user_pid){

        int user_tp_id =0;
        struct task_struct *p;

        int my_tty =0;

        // enumerate all the processes
        for_each_process(p) {
```

```
                //printk(KERN_CRIT "PID is %d and TP_ID is %d \n", p->pid, p->tp_id);

                // identify the process user_pid
                if(p->pid == user_pid){

                        //get the trusted path id of the user_pid
                        user_tp_id = p->tp_id;
                        my_tty = (int) p->tty;
                }
        }

        // test to make sure that we did find a user_tp_id
        if (!user_tp_id) return -1;

        // user_tp_id now contains the correct number
        // user_pid is the trusted path pid
        for_each_process(p){

                // if p has the correct tp_id
                if(p->tp_id == user_tp_id){
                        //printk(KERN_CRIT "Bingo on pid %d\n", p->pid);
                        suspend_task(p);

                }
        }

        // test routine to visualize the status of all the processes
        /*{
                printk(KERN_CRIT "Suspend debug routine\n");
                for_each_process(p){
                //      int my_rq;
                //      my_rq = task_rq(p);
                        printk(KERN_CRIT "pid=%d   do_not_activate=%d \n",
                                p->pid, p->do_not_activate);
                }
        }*/
        return 0;
}

// Takes an int that is the pid of the trusted path
// gets the tp_id and restores all processes with that tp_id
// except the trusted path
int restore_all_processes( user_pid){
```

```
	int user_tp_id =0;
	struct task_struct *p;

	   // find the tp_id of user_pid
	for_each_process(p) {

	      // printk(KERN_CRIT "PID is %d and TP_ID is %d \n",
	                  //p->pid, p->tp_id);
	       if(p->pid == user_pid){
	            user_tp_id = p->tp_id;
	            //printk(KERN_CRIT "user tp id is %d\n", user_tp_id);
	       }
	   }

	   //test to make sure we were able to find the tp_id
	   if(! user_tp_id)
	            return -1;

	   // restore all the processes with that tp_id
	   for_each_process(p) {

	            // if p has the correct tp_id
	      if(p->tp_id == user_tp_id){
	                  resume_task(p);
	      }
	}

	// test routine to visualize the status of all the processes
	/* {
	      printk(KERN_CRIT "Restore debug routine\n");
	      for_each_process(p){
	      //    int my_rq;
	      //    my_rq = task_rq(p);
	            printk(KERN_CRIT   "pid=%d       do_not_activate=%d  \n",  p->pid,  p-
>do_not_activate);
	      }
	}*/

	return 0;
}

// helper function to kill all the processes associated
// a session
void  kill_all_processes(int user_pid){
```

```
        int user_tp_id =0;
        struct task_struct *p;

        // enumerate all the processes
        for_each_process(p) {

                // identify the process user_pid
            if(p->pid == user_pid){
                //get the trusted path id of the user_pid
                user_tp_id = p->tp_id;
                //printk(KERN_CRIT "user tp id is %d\n", user_tp_id);
            }
        }
           // terminate each process with the tp_id
           for_each_process(p) {

                //printk(KERN_CRIT "PID is %d tp_id is %d\n",
                //      p->pid, p->tp_id);
            // if p has the correct tp_id
            if(p->tp_id == user_tp_id){

                        // printk(KERN_CRIT "Bingo on pid %d\n", p->pid);
                        printk(KERN_CRIT "terminating pid %d\n", p->pid);
                        force_sig(SIGKILL, p);
            }
        }
}

asmlinkage int sys_trustedpath(int type_msg, int  pid){

           printk(KERN_CRIT "Pid %d called tp\n", current->pid);

           switch (type_msg) {

                case REGISTER:
                        // set the lock after the first registration
                        // there could be some potential problems
                        // e.g. if the trusted path dies we would
                        // have to reboot
                        if (!lock){
                                tpum = pid;
                                lock = 1;
                                printk(KERN_CRIT
                                        "Trusted path registerd for pid %i\n"
                                        , pid);
```

```
                }
                break;

        case SUSPEND:
                if (current->pid == tpum){
                        suspend_all_processes(pid);
                        printk(KERN_CRIT "tp suspend\n");
                }
            break;

        case RESTORE:
                if (current->pid == tpum){
                        restore_all_processes(pid);
                        printk(KERN_CRIT "tp restore\n");
                }
            break;

        case KILL:
                if (current->pid == tpum){
                        kill_all_processes(pid);
                        printk(KERN_CRIT "tp kill\n");
                }
            break;

            default:
                printk(KERN_CRIT "incorrect trusted path syscall\n");
    }

    return(0);
}

EXPORT_SYMBOL(handle_trustedpath);
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. TRUSTED PATH SYSTEM CALL

This appendix provides detailed implementation information for the trusted path system call.

Adding the interrupt

In the arch/i386/kernel/entry.S file add at line 1040

```
/* system call for the trusted path
 * ahilchie 7-12-04
 */
.long sys_trustedpath            /* 274 */
```

Adding the stub

In the include/asm-i386/unistd.h file add at line 282

```
#define __NR_trustedpath      274
```

at line 284 increment the number of valid syscalls

```
#define NR_syscalls           275
```

Modifications to trustedpath.h

Added access to the system call macro file.

```
#include <linux/linkage.h>
#define REGISTER 0
#define SUSPEND 1
#define RESTORE 2
#define KILL 3
```

Modifications to trustedpath.c

```
/* handle trusted path system calls */

asmlinkage int sys_trustedpath( int type_msg, int user_pid){

        switch (type_msg) {
                case REGISTER:
                        …
                        break;
                case SUSPEND:
                        …
```

```
                    break;
                case RESTORE:
                    …
                    break;
                case KILL:
                    …
                    break;
                default:
                    …
        }
        return(0);
}
```

User space header to support the system call

Userspace.h

```
#define REGISTER 0
#define SUSPEND 1
#define RESTORE 2
#define KILL 3

#include <linux/unistd.h>

_syscall2(int, trustedpath, int, type_msg, int, user_pid);
```

APPENDIX D. SCHEDULE MODIFICATION

This appendix contains the modifications made to the sched.h and sched.c files in the kernel source tree. Code added as part of this work is in bold face.

sched.h

Due to the size of the original files, only the process table is presented.

The process table, *task_struct,* was modified to include a new integer process ID for use by the trusted path. This is labeled as tp_id and is used to determine the original parent of processes.

```
struct task_struct {
        volatile long state; /*-1 unrunnable, 0 runnable, >0 stopped*/
        struct thread_info *thread_info;
        atomic_t usage;
        unsigned long flags;    /* per process flags, defined below */
        unsigned long ptrace;

        int lock_depth;                 /* Lock depth */

        int prio, static_prio;
        struct list_head run_list;
        prio_array_t *array;
/* DJS - Flag to control the activation of suspended tasks */
        int do_not_activate;

        unsigned long sleep_avg;
        long interactive_credit;
        unsigned long long timestamp;
        int activated;

        unsigned long policy;
        cpumask_t cpus_allowed;
        unsigned int time_slice, first_time_slice;

        struct list_head tasks;
        struct list_head ptrace_children;
        struct list_head ptrace_list;

        struct mm_struct *mm, *active_mm;
```

```
/* task state */
        struct linux_binfmt *binfmt;
        int exit_code, exit_signal;
        int pdeath_signal;  /*  The signal sent when the parent dies  */
        /* ??? */
        unsigned long personality;
        int did_exec:1;
        pid_t pid;
        pid_t __pgrp;           /* Accessed via process_group() */
        pid_t tty_old_pgrp;
        pid_t session;
        pid_t tgid;

        // trusted path id -- tp_id
        // ahilchie 6-16-04
        //  the true parent pid of this process
        //  to allow finding all the processes associated with a user

        pid_t tp_id;

        /* boolean value for session group leader */
        int leader;
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->parent->pid)
         */
        struct task_struct *real_parent; /* real parent process (when being
debugged) */
        struct task_struct *parent;     /* parent process */
        struct list_head children;      /* list of my children */
        struct list_head sibling;       /* linkage in my parent's children list */
        struct task_struct *group_leader;       /* threadgroup leader */

        /* PID/PID hash table linkage. */
        struct pid_link pids[PIDTYPE_MAX];

        wait_queue_head_t wait_chldexit;    /* for wait4() */
        struct completion *vfork_done;              /* for vfork() */
        int __user *set_child_tid;              /* CLONE_CHILD_SETTID */
        int __user *clear_child_tid;            /* CLONE_CHILD_CLEARTID */

        unsigned long rt_priority;
        unsigned long it_real_value, it_prof_value, it_virt_value;
        unsigned long it_real_incr, it_prof_incr, it_virt_incr;
```

```
        struct timer_list real_timer;
        struct list_head posix_timers; /* POSIX.1b Interval Timers */
        unsigned long utime, stime, cutime, cstime;
        unsigned long nvcsw, nivcsw, cnvcsw, cnivcsw; /* context switch counts
*/
        u64 start_time;
/* mm fault and swap info: this can arguably be seen as either mm-specific or
thread-specific */
        unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        struct group_info *group_info;
        kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
        int keep_capabilities:1;
        struct user_struct *user;
/* limits */
        struct rlimit rlim[RLIM_NLIMITS];
        unsigned short used_math;
        char comm[16];
/* file system info */
        int link_count, total_link_count;
        struct tty_struct *tty; /* NULL if no tty */
/* ipc stuff */
        struct sysv_sem sysvsem;
/* CPU-specific state of this task */
        struct thread_struct thread;
/* filesystem information */
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
/* namespace */
        struct namespace *namespace;
/* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;

        sigset_t blocked, real_blocked;
        struct sigpending pending;

        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;
```

```
        /* TUX state */
        void *tux_info;
        void (*tux_exit)(void);


        void *security;
        struct audit_context *audit_context;

    /* Thread group tracking */
        u32 parent_exec_id;
        u32 self_exec_id;
    /* Protection of (de-)allocation: mm, files, fs, tty */
        spinlock_t alloc_lock;
    /* Protection of proc_dentry: nesting proc_lock, dcache_lock,
    write_lock_irq(&tasklist_lock); */
        spinlock_t proc_lock;
    /* context-switch lock */
        spinlock_t switch_lock;

    /* journalling filesystem info */
        void *journal_info;

    /* VM state */
        struct reclaim_state *reclaim_state;

        struct dentry *proc_dentry;
        struct backing_dev_info *backing_dev_info;

        struct io_context *io_context;

        unsigned long ptrace_message;
        siginfo_t *last_siginfo; /* For ptrace use.  */
        };
```

```
// ahilchie trusted path mod by DJS
void suspend_task(struct task_struct *p);
void resume_task(struct task_struct *p);
```

sched.c

Only the functions that were modified are presented

```
struct runqueue {
    spinlock_t lock;
    unsigned long long nr_switches;
    unsigned long nr_running, expired_timestamp, nr_uninterruptible,
```

```
                timestamp_last_tick;
/* DJS - keep track of how many tasks are suspended */
        unsigned long nr_suspend;
        task_t *curr, *idle;
        struct mm_struct *prev_mm;
        prio_array_t *active, *expired, arrays[2];
/* DJS - Here is where we keep the suspended tasks */
        prio_array_t *suspend_active,*suspend_expired, suspend_arrays[2];
        int best_expired_prio, prev_cpu_load[NR_CPUS];
#ifdef CONFIG_NUMA
        atomic_t *node_nr_running;
        int prev_node_load[MAX_NUMNODES];
#endif
        task_t *migration_thread;
        struct list_head migration_queue;

        atomic_t nr_iowait;
};

/* DJS - macros for handling the nr_suspend counters */
/* DJS - Hhmmm - why wasn't nr_running initiailzed to zero? */
# define nr_suspend_init(rq)        do { (rq)->nr_suspend=0; } while (0)
# define nr_suspend_inc(rq)         do { (rq)->nr_suspend++; } while (0)
# define nr_suspend_dec(rq)         do { (rq)->nr_suspend--; } while (0)

/*
 * __activate_task - move a task to the runqueue.
 */
static inline void __activate_task(task_t *p, runqueue_t *rq)
{
/* DJS - Only activate if were allowed (e.g. it is not suspended) */
        if (!p->do_not_activate) {
                enqueue_task(p, rq->active);
                nr_running_inc(rq);
        }
}

/* DJS - New function to suspend a task */
/*
 * suspend_task - move a task from the active/expired arrays
 * to the suspend_active/suspend_expire arrays
 */
//static inline
void suspend_task(struct task_struct *p) //, runqueue_t *rq)
{
```

```
        int is_active=1;

        // ahilchie added
        struct runqueue *rq;
        rq = task_rq(p);

        if(p->array)
        {

/* DJS - determine if the task is active or expired */
                if (p->array == rq->expired)
                        is_active=0;
                nr_running_dec(rq);

                dequeue_task(p, p->array);
                if (is_active) {
                        enqueue_task(p, rq->suspend_active);
                }else{
                        enqueue_task(p, rq->suspend_expired);
                }
                nr_suspend_inc(rq);
        }else{
                // Mark the task so it doesn't get activated
                p->do_not_activate = 1;
        }
}

/* DJS - New function to resume a suspended a task */
/*
 * resume_task - move a task from the suspend_active/suspend_expired arrays
 * to the active/expire arrays
 */
//static inline
void resume_task(struct task_struct *p)//, runqueue_t *rq)
{
        int is_active=1;
      // ahilchie added
      struct runqueue *rq;
      rq = task_rq(p);

        if(p->array)
        {
/* DJS - determine if the task is active or expired */
                if (p->array == rq->suspend_expired)
                        is_active=0;
```

```
                nr_suspend_dec(rq);
                dequeue_task(p, p->array);
                if (is_active) {
                        enqueue_task(p, rq->active);
                }else{
                        enqueue_task(p, rq->expired);
                }
                nr_running_inc(rq);
        }else{
                // Mark the task so it can get activated
                p->do_not_activate = 0;
        }
}

static int try_to_wake_up(task_t * p, unsigned int state, int sync)
{
        unsigned long flags;
        int success = 0;
        long old_state;
        runqueue_t *rq;

repeat_lock_task:
        rq = task_rq_lock(p, &flags);
        old_state = p->state;
        if (old_state & state) {
                if (!p->array) {
                        /*
                         * Fast-migrate the task if it's not running or runnable
                         * currently. Do not violate hard affinity.
                         */
                        if (unlikely(sync && !task_running(rq, p) &&
                                (task_cpu(p) != smp_processor_id()) &&
                                        cpu_isset(smp_processor_id(),
                                                        p->cpus_allowed) &&
                                        !cpu_is_offline(smp_processor_id()))) {
                                set_task_cpu(p, smp_processor_id());
                                task_rq_unlock(rq, &flags);
                                goto repeat_lock_task;
                        }
                        if (old_state == TASK_UNINTERRUPTIBLE) {
                                rq->nr_uninterruptible--;
                                /*
                                 * Tasks on involuntary sleep don't earn
                                 * sleep_avg beyond just interactive state.
                                 */
```

```
                        p->activated = -1;
                }
                if (sync && (task_cpu(p) == smp_processor_id()))
                        __activate_task(p, rq);
                else {
                        activate_task(p, rq);
                        if (TASK_PREEMPTS_CURR(p, rq))
                                resched_task(rq->curr);
                }
                success = 1;
        }
/* DJS - Only change the state if it is in a runqueue */
        if (p->array) {
                p->state = TASK_RUNNING;
        }
    }
    task_rq_unlock(rq, &flags);

    return success;
}

void fastcall wake_up_forked_process(task_t * p)
{
    unsigned long flags;
    runqueue_t *rq = task_rq_lock(current, &flags);

    BUG_ON(p->state != TASK_RUNNING);

    /*
     * We decrease the sleep average of forking parents
     * and children as well, to keep max-interactive tasks
     * from forking tasks that are max-interactive.
     */
    current->sleep_avg = JIFFIES_TO_NS(CURRENT_BONUS(current) *
        PARENT_PENALTY / 100 * MAX_SLEEP_AVG / MAX_BONUS);

    p->sleep_avg = JIFFIES_TO_NS(CURRENT_BONUS(p) *
        CHILD_PENALTY / 100 * MAX_SLEEP_AVG / MAX_BONUS);

    p->interactive_credit = 0;

    p->prio = effective_prio(p);
    set_task_cpu(p, smp_processor_id());

    if (unlikely(!current->array))
```

```
                __activate_task(p, rq);
        else {
/* DJS - Only add it, if it is not suspended */
                if (!p->do_not_activate) {
                        p->prio = current->prio;
                        list_add_tail(&p->run_list, &current->run_list);
                        p->array = current->array;
                        p->array->nr_active++;
                        nr_running_inc(rq);
                }
        }
        task_rq_unlock(rq, &flags);
}


void __init sched_init(void)
{
        runqueue_t *rq;
        int i, j, k;

        for (i = 0; i < NR_CPUS; i++) {
                prio_array_t *array;

                rq = cpu_rq(i);
                rq->active = rq->arrays;
                rq->expired = rq->arrays + 1;
/* DJS - init the suspend arrays */
                rq->suspend_active = rq->suspend_arrays;
                rq->suspend_expired = rq->suspend_arrays + 1;
                rq->best_expired_prio = MAX_PRIO;

                spin_lock_init(&rq->lock);
                INIT_LIST_HEAD(&rq->migration_queue);
                atomic_set(&rq->nr_iowait, 0);
                nr_running_init(rq);

                for (j = 0; j < 2; j++) {
                        array = rq->arrays + j;
                        for (k = 0; k < MAX_PRIO; k++) {
                                INIT_LIST_HEAD(array->queue + k);
                                __clear_bit(k, array->bitmap);
                        }
                        // delimiter for bitsearch
                        __set_bit(MAX_PRIO, array->bitmap);
                }
```

```
/* DJS - more init of the suspend arrays */
                nr_suspend_init(rq);
                for (j = 0; j < 2; j++) {
                        array = rq->suspend_arrays + j;
                        for (k = 0; k < MAX_PRIO; k++) {
                                INIT_LIST_HEAD(array->queue + k);
                                __clear_bit(k, array->bitmap);
                        }
                        // delimiter for bitsearch
                        __set_bit(MAX_PRIO, array->bitmap);
                }
        }
        /*
         * We have to do a little magic to get the first
         * thread right in SMP mode.
         */
        rq = this_rq();
        rq->curr = current;
        rq->idle = current;
/* DJS - make sure we can activate it */
        current->do_not_activate = 0;
        set_task_cpu(current, smp_processor_id());
        wake_up_forked_process(current);

        init_timers();

        /*
         * The boot idle thread does lazy MMU switching as well:
         */
        atomic_inc(&init_mm.mm_count);
        enter_lazy_tlb(&init_mm, current);
}
```

APPENDIX E. KEYBOARD DRIVER MODIFICATIONS

This appendix contains the modification to the keyboard driver for the SELinux trusted path. For the sake of understandability the entire kbd_keycode function, which contains the additional trusted path code, is provided. Code added as part of this work is in bold face.

```
void kbd_keycode(unsigned int keycode, int down, struct pt_regs *regs)
{
        struct vc_data *vc = vc_cons[fg_console].d;
        unsigned short keysym, *key_map;
        unsigned char type, raw_mode;
        struct tty_struct *tty;
        int shift_final;

        if (down != 2)
                add_keyboard_randomness((keycode << 1) ^ down);

        tty = vc->vc_tty;

        if (tty && (!tty->driver_data)) {
                /* No driver data? Strange. Okay we fix it then. */
                tty->driver_data = vc;
        }

        kbd = kbd_table + fg_console;

        if (keycode == KEY_LEFTALT || keycode == KEY_RIGHTALT)
                sysrq_alt = down;
#if defined(CONFIG_SPARC32) || defined(CONFIG_SPARC64)
        if (keycode == KEY_STOP)
                sparc_l1_a_state = down;
#endif

        rep = (down == 2);

#ifdef CONFIG_MAC_EMUMOUSEBTN
        if (mac_hid_mouse_emulate_buttons(1, keycode, down))
                return;
#endif /* CONFIG_MAC_EMUMOUSEBTN */

        if ((raw_mode = (kbd->kbdmode == VC_RAW)))
```

```
                        if (emulate_raw(vc, keycode, !down << 7))
                                if (keycode < BTN_MISC)
                                        printk(KERN_WARNING     "keyboard.c:     can't
emulate rawmode for keycode %d\n", keycode);
        /
         * ahilchie trusted path 6-13-0
         *
         * catch the trusted path SAK and notify the
         * kernel trusted path modul

         *
             if (keycode == KEY_LEFTCTRL || keycode == KEY_RIGHTCTRL)
                 tp_ctrl = down;

               if(keycode == KEY_PAUSE && tp_ctrl ){
                        handle_trustedpath(regs, tty);
                        return;
               }


        #ifdef CONFIG_MAGIC_SYSRQ          /* Handle the SysRq Hack */
                if (keycode == KEY_SYSRQ && (sysrq_down || (down == 1 &&
sysrq_alt))) {
                        sysrq_down = down;
                        return;
                }
                if (sysrq_down && down && !rep) {
                        handle_sysrq(kbd_sysrq_xlate[keycode], regs, tty);
                        return;
                }
        #endif
        #if defined(CONFIG_SPARC32) || defined(CONFIG_SPARC64)
                if (keycode == KEY_A && sparc_l1_a_state) {
                        sparc_l1_a_state = 0;
                        sun_do_break();
                }
        #endif

                if (kbd->kbdmode == VC_MEDIUMRAW) {
                        /*
                         * This is extended medium raw mode, with keys above 127
                         * encoded as 0, high 7 bits, low 7 bits, with the 0 bearing
                         * the 'up' flag if needed. 0 is reserved, so this shouldn't
                         * interfere with anything else. The two bytes after 0 will
                         * always have the up flag set not to interfere with older
```

```
		 * applications. This allows for 16384 different keycodes,
		 * which should be enough.
		 */
		if (keycode < 128) {
			put_queue(vc, keycode | (!down << 7));
		} else {
			put_queue(vc, !down << 7);
			put_queue(vc, (keycode >> 7) | 0x80);
			put_queue(vc, keycode | 0x80);
		}
		raw_mode = 1;
}

if (down)
		set_bit(keycode, key_down);
else
		clear_bit(keycode, key_down);

if (rep && (!vc_kbd_mode(kbd, VC_REPEAT) || (tty &&
		(!L_ECHO(tty) && tty->driver->chars_in_buffer(tty))))) {
		/*
		 * Don't repeat a key if the input buffers are not empty and the
		 * characters get aren't echoed locally. This makes key repeat
		 * usable with slow applications and under heavy loads.
		 */
		return;
}

shift_final = (shift_state | kbd->slockstate) ^ kbd->lockstate;
key_map = key_maps[shift_final];

if (!key_map) {
		compute_shiftstate();
		kbd->slockstate = 0;
		return;
}

keysym = key_map[keycode];
type = KTYP(keysym);

if (type < 0xf0) {
		if (down && !raw_mode) to_utf8(vc, keysym);
		return;
}
```

```
        type -= 0xf0;

        if (raw_mode && type != KT_SPEC && type != KT_SHIFT)
                return;

        if (type == KT_LETTER) {
                type = KT_LATIN;
                if (vc_kbd_led(kbd, VC_CAPSLOCK)) {
                        key_map = key_maps[shift_final ^ (1 << KG_SHIFT)];
                        if (key_map)
                                keysym = key_map[keycode];
                }
        }

        (*k_handler[type])(vc, keysym & 0xff, !down, regs);

        if (type != KT_SLOCK)
                kbd->slockstate = 0;
}
```

APPENDIX F. FORK MODIFICATION

This appendix contains the modification made to the fork.c file in the kernel source tree. Code added as part of this work is in bold face. Due to the size of the original files, only the process copy, *copy_process,* function is presented. The *copy process* is called during the creation of all processes except for the original process *init*.

The modification sets the trusted path identifier, *tp_id,* to the *tp_id* of its parent process unless it's parent is *init*, e.g., process ID 1, in which case the tp_id is set to the current process ID. Since *init* will start the *getty* process each time each session has a single unique *tp_id* that is not modified during the course of execution.

```
struct task_struct *copy_process(unsigned long clone_flags,
                                 unsigned long stack_start,
                                 struct pt_regs *regs,
                                 unsigned long stack_size,
                                 int __user *parent_tidptr,
                                 int __user *child_tidptr)
{
        int retval;
        struct task_struct *p = NULL;

        if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) ==
(CLONE_NEWNS|CLONE_FS))
                return ERR_PTR(-EINVAL);

        /*
         * Thread groups must share signals as well, and detached threads
         * can only be started up within the thread group.
         */
        if ((clone_flags & CLONE_THREAD) && !(clone_flags &
CLONE_SIGHAND))
                return ERR_PTR(-EINVAL);

        /*
         * Shared signal handlers imply shared VM. By way of the above,
         * thread groups also imply shared VM. Blocking this case allows
         * for various simplifications in other code.
         */
        if ((clone_flags & CLONE_SIGHAND) && !(clone_flags &
CLONE_VM))
```

```
                return ERR_PTR(-EINVAL);

        retval = security_task_create(clone_flags);
        if (retval)
                goto fork_out;

        retval = -ENOMEM;
        p = dup_task_struct(current);
        if (!p)
                goto fork_out;
        p->tux_info = NULL;

        retval = -EAGAIN;
        if (atomic_read(&p->user->processes) >=
                        p->rlim[RLIMIT_NPROC].rlim_cur) {
                if                (!capable(CAP_SYS_ADMIN)                &&
!capable(CAP_SYS_RESOURCE) &&
                                p->user != &root_user)
                        goto bad_fork_free;
        }

        atomic_inc(&p->user->__count);
        atomic_inc(&p->user->processes);
        get_group_info(p->group_info);

        /*
         * If multiple threads are within copy_process(), then this check
         * triggers too late. This doesn't hurt, the check is only there
         * to stop root fork bombs.
         */
        if (nr_threads >= max_threads)
                goto bad_fork_cleanup_count;

        if (!try_module_get(p->thread_info->exec_domain->module))
                goto bad_fork_cleanup_count;

        if (p->binfmt && !try_module_get(p->binfmt->module))
                goto bad_fork_cleanup_put_domain;

        p->did_exec = 0;
        copy_flags(clone_flags, p);
        if (clone_flags & CLONE_IDLETASK)
                p->pid = 0;
        else {
                p->pid = alloc_pidmap();
```

```
            if (p->pid == -1)
                    goto bad_fork_cleanup;
    }
    retval = -EFAULT;
    if (clone_flags & CLONE_PARENT_SETTID)
            if (put_user(p->pid, parent_tidptr))
                    goto bad_fork_cleanup;

    p->proc_dentry = NULL;

    INIT_LIST_HEAD(&p->children);
    INIT_LIST_HEAD(&p->sibling);
    INIT_LIST_HEAD(&p->posix_timers);
    init_waitqueue_head(&p->wait_chldexit);
    p->vfork_done = NULL;
    spin_lock_init(&p->alloc_lock);
    spin_lock_init(&p->proc_lock);

    clear_tsk_thread_flag(p, TIF_SIGPENDING);
    init_sigpending(&p->pending);

    p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
    p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
    init_timer(&p->real_timer);
    p->real_timer.data = (unsigned long) p;

    p->leader = 0;            /* session leadership doesn't inherit */
    p->tty_old_pgrp = 0;
    p->utime = p->stime = 0;
    p->cutime = p->cstime = 0;
    p->lock_depth = -1;             /* -1 = no lock */
    p->start_time = get_jiffies_64();
    p->security = NULL;
    p->io_context = NULL;
p->audit_context = NULL;

    retval = -ENOMEM;
    if ((retval = security_task_alloc(p)))
            goto bad_fork_cleanup;
if ((retval = audit_alloc(p)))
        goto bad_fork_cleanup_security;
    /* copy all the process information */
    if ((retval = copy_semundo(clone_flags, p)))
            goto bad_fork_cleanup_audit;
    if ((retval = copy_files(clone_flags, p)))
```

```
                goto bad_fork_cleanup_semundo;
        if ((retval = copy_fs(clone_flags, p)))
                goto bad_fork_cleanup_files;
        if ((retval = copy_sighand(clone_flags, p)))
                goto bad_fork_cleanup_fs;
        if ((retval = copy_signal(clone_flags, p)))
                goto bad_fork_cleanup_sighand;
        if ((retval = copy_mm(clone_flags, p)))
                goto bad_fork_cleanup_signal;
        if ((retval = copy_namespace(clone_flags, p)))
                goto bad_fork_cleanup_mm;
        retval = copy_thread(0, clone_flags, stack_start, stack_size, p, regs);
        if (retval)
                goto bad_fork_cleanup_namespace;

        p->set_child_tid = (clone_flags & CLONE_CHILD_SETTID) ?
child_tidptr : NULL;
        /*
         * Clear TID on mm_release()?
         */
        p->clear_child_tid = (clone_flags & CLONE_CHILD_CLEARTID) ?
child_tidptr: NULL;

        /*
         * Syscall tracing should be turned off in the child regardless
         * of CLONE_PTRACE.
         */
        clear_tsk_thread_flag(p, TIF_SYSCALL_TRACE);

        /* Our parent execution domain becomes current domain
           These must match for thread signalling to apply */

        p->parent_exec_id = p->self_exec_id;

        /* ok, now we should be set up.. */
        p->exit_signal = (clone_flags & CLONE_THREAD) ? -1 : (clone_flags &
CSIGNAL);
        p->pdeath_signal = 0;

        /* Perform scheduler related setup */
        sched_fork(p);

        /*
         * Ok, make it visible to the rest of the system.
         * We dont wake it up yet.
```

```
 */
p->tgid = p->pid;
p->group_leader = p;
INIT_LIST_HEAD(&p->ptrace_children);
INIT_LIST_HEAD(&p->ptrace_list);

/* Need tasklist lock for parent etc handling! */
write_lock_irq(&tasklist_lock);
/*
 * Check for pending SIGKILL! The new thread should not be allowed
 * to slip out of an OOM kill. (or normal SIGKILL.)
 */
if (sigismember(&current->pending.signal, SIGKILL)) {
        write_unlock_irq(&tasklist_lock);
        retval = -EINTR;
        goto bad_fork_cleanup_namespace;
}

/* CLONE_PARENT re-uses the old parent */
if (clone_flags & CLONE_PARENT)
        p->real_parent = current->real_parent;
else
        p->real_parent = current;
p->parent = p->real_parent;

/* tp_id set to real parent
// ahilchie 6-16-04
// to allow finding all the processes associated with a user
// this avoids the CLONE_PARENT logic above
// and gives us a legitimate tp_id

// The logic
//
// first pid > 0 will be the trusted path
// the next one will be the shell pid

// if the parent of the current (parent)
// process is init e.g.
// init is 0 which spawns
// trusted path (current) is XXX which spawns
// session shell (p)  is XXX + X
if ( ! current->parent->parent->pid )
        //then the tp_id is p->pid
        p->tp_id = p->pid;
else
```

```
        // if init is not the grandparent
        // then use parents tp_id
        p->tp_id = current->tp_id;

if (clone_flags & CLONE_THREAD) {
        spin_lock(&current->sighand->siglock);
        /*
         * Important: if an exit-all has been started then
         * do not create this new thread - the whole thread
         * group is supposed to exit anyway.
         */
        if (current->signal->group_exit) {
                spin_unlock(&current->sighand->siglock);
                write_unlock_irq(&tasklist_lock);
                retval = -EAGAIN;
                goto bad_fork_cleanup_namespace;
        }
        p->tgid = current->tgid;
        p->group_leader = current->group_leader;

        if (current->signal->group_stop_count > 0) {
                /*
                 * There is an all-stop in progress for the group.
                 * We ourselves will stop as soon as we check signals.
                 * Make the new thread part of that group stop too.
                 */
                current->signal->group_stop_count++;
                set_tsk_thread_flag(p, TIF_SIGPENDING);
        }

        spin_unlock(&current->sighand->siglock);
}

SET_LINKS(p);
if (p->ptrace & PT_PTRACED)
        __ptrace_link(p, current->parent);

attach_pid(p, PIDTYPE_PID, p->pid);
if (thread_group_leader(p)) {
        attach_pid(p, PIDTYPE_TGID, p->tgid);
        attach_pid(p, PIDTYPE_PGID, process_group(p));
        attach_pid(p, PIDTYPE_SID, p->session);
        if (p->pid)
                __get_cpu_var(process_counts)++;
} else
```

```
                link_pid(p, p->pids + PIDTYPE_TGID, &p->group_leader->pids[PIDTYPE_TGID].pid);

        nr_threads++;
        write_unlock_irq(&tasklist_lock);
        retval = 0;

fork_out:
        if (retval)
                return ERR_PTR(retval);
        return p;

bad_fork_cleanup_namespace:
        exit_namespace(p);
bad_fork_cleanup_mm:
        exit_mm(p);
bad_fork_cleanup_signal:
        exit_signal(p);
bad_fork_cleanup_sighand:
        exit_sighand(p);
bad_fork_cleanup_fs:
        exit_fs(p); /* blocking */
bad_fork_cleanup_files:
        exit_files(p); /* blocking */
bad_fork_cleanup_semundo:
        exit_sem(p);
bad_fork_cleanup_audit:
        audit_free(p);
bad_fork_cleanup_security:
        security_task_free(p);
bad_fork_cleanup:
        if (p->pid > 0)
                free_pidmap(p->pid);
        if (p->binfmt)
                module_put(p->binfmt->module);
bad_fork_cleanup_put_domain:
        module_put(p->thread_info->exec_domain->module);
bad_fork_cleanup_count:
        put_group_info(p->group_info);
        atomic_dec(&p->user->processes);
        free_uid(p->user);
bad_fork_free:
        free_task(p);
        goto fork_out;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G. INSTALLATION GUIDE

Installation of Fedora Core 2.2

Boot to CDROM FC 2.2 disk 1
Graphical install
Skip CDROM check
Normal selections e.g. English
Custom install
Automatic partitioning with defaults
Default grup loader
Default DHCP
Default firewall
Active SELinux extensions
Package selection (just changes from defaults are listed)
- Editors selected
- No sound or video
- No graphics
- Check all development boxes except KDE
- Administrative tools
- System tools
- No printing support

Edit /etc/inittab to start to at runlevel 3

Installation of SELinux Policy Files

Rpm –ivh checkpolicy-1-8-1.i386.rpm disk 3
Rpm –ivh policy-sources-1.9.15.noarch.rpm disk 3

Change the policy files as indicated in Appendix H.

Additionally comment out or remove the following two lines from assert.te
Line # 44 neverallow { domain -auth -auth_write } shadow_t:file ~getattr;
Line # 115 ifdef(`getty.te', `assert_execute(getty)')

Rebuild policy by cd /etc/security/selinux/src/policy and using the make install command.

Linux Configuration

Remove linux services such as sendmail, and all the other unneeded default services.

Trusted Path Kernel Configuration

Install the trusted path kernel files

Filename	Install location from root of source tree
Trustedpath.h	include/linux/
Trustedpath.c	drivers/char/
Fork.c	kernel/
Keyboard.c	drivers/char/
Keyboard.h	drivers/char/
Sched.h	include/linux/
Sched.c	kernel/

Modify the make file in the drivers/char/ directory to include the trustedpath files e.g. change the following line in the Makefile:
obj-y += mem.o random.o tty_io.o n_tty.o tty_ioctl.o pty.o misc.o
to read:
obj-y += mem.o random.o tty_io.o n_tty.o tty_ioctl.o pty.o misc.o ***trustedpath.o***

Make the trusted path system call changes as indicated in Appendix C.

Rebuild kernel

Make xconfig – remove objects that are not needed. As an example, remove pcmcia support.

Make
Echo 0 > /selinux/enforce to temporarily disable SELinux
Make modules_install
Make install

Modify the boot loader to automatically boot to the new kernel

Install the login files

Copy the util-login.tgz file to a directory and untar it. Move into the util/login-utils/ directory and issue the make command. Rename the login file as tp_getty. Move tp_getty to the desired location and perform the command chcon system_u:object_r:login_exec_t tp_getty. This command will set the selinux context of the file correctly.

APPENDIX H. SE LINUX POLICY CONFIGURATION

This appendix provides the modified policy files that were created for this project. The modifications are in bold.

The login.te file

Typically located in /etc/security/selinux/policy/domains/programs.

```
#DESC Login - Local/remote login utilities
#
# Authors:  Stephen Smalley <sds@epoch.ncsc.mil> and Timothy Fraser
# Macroised by Russell Coker <russell@coker.com.au>
# X-Debian-Packages: login
#

###################################
#
# Rules for the local_login_t domain
# and the remote_login_t domain.
#

# $1 is the name of the domain (local or remote)
# I added "mlstrustedreader, mlstrustedwriter, mlstrustedobject" to
# remote_login_t, not sure if this is right
define(`login_domain', `
type $1_login_t, domain, privuser, privrole, privlog, auth_chkpwd, privowner,
mlstrustedreader, mlstrustedwriter, mlstrustedobject, privfd;
role system_r types $1_login_t;

dontaudit $1_login_t shadow_t:file { getattr read };

general_domain_access($1_login_t);

# Read system information files in /proc.
allow $1_login_t proc_t:dir r_dir_perms;
allow $1_login_t proc_t:notdevfile_class_set r_file_perms;

base_file_read_access($1_login_t)

# Read directories and files with the readable_t type.
# This type is a general type for "world"-readable files.
allow $1_login_t readable_t:dir r_dir_perms;
allow $1_login_t readable_t:notdevfile_class_set r_file_perms;
```

```
# Read /var, /var/spool
allow $1_login_t { var_t var_spool_t }:dir search;

# for when /var/mail is a sym-link
allow $1_login_t var_t:lnk_file read;

# Read /etc.
allow $1_login_t etc_t:dir r_dir_perms;
allow $1_login_t etc_t:notdevfile_class_set r_file_perms;
allow $1_login_t etc_runtime_t:{ file lnk_file } r_file_perms;

read_locale($1_login_t)

# for SSP/ProPolice
allow $1_login_t urandom_device_t:chr_file { getattr read };

# Read executable types.
allow $1_login_t exec_type:{ file lnk_file } r_file_perms;

# Read /dev directories and any symbolic links.
allow $1_login_t device_t:dir r_dir_perms;
allow $1_login_t device_t:lnk_file r_file_perms;

uses_shlib($1_login_t);

tmp_domain($1_login)

ifdef(`pam.te', `
can_exec($1_login_t, pam_exec_t)
')

# Use capabilities
allow $1_login_t self:capability { dac_override chown fowner fsetid kill setgid
setuid net_bind_service sys_nice sys_resource sys_tty_config };

# Set exec context.
can_setexec($1_login_t)

ifdef(`automount.te', `
allow $1_login_t autofs_t:dir { search };
')
allow $1_login_t mnt_t:dir r_dir_perms;

ifdef(`nfs_home_dirs', `
```

```
r_dir_file($1_login_t, nfs_t)
')dnl end if nfs_home_dirs

#
# /var/run/console requires the following
#
ifdef(`xdm.te', `
create_dir_file($1_login_t, xdm_var_run_t)
allow xdm_t $1_login_t:process { signull };
')

# Permit login to search the user home directories.
allow $1_login_t home_root_t:dir search;
allow $1_login_t home_dir_type:dir search;

# Write to /var/run/utmp.
allow $1_login_t var_run_t:dir search;
allow $1_login_t initrc_var_run_t:file rw_file_perms;

# Write to /var/log/wtmp.
allow $1_login_t var_log_t:dir search;
allow $1_login_t wtmp_t:file rw_file_perms;

# Write to /var/log/lastlog.
allow $1_login_t lastlog_t:file rw_file_perms;

# Write to /var/log/btmp
allow $1_login_t faillog_t:file { append read write };

# Search for mail spool file.
allow $1_login_t mail_spool_t:dir r_dir_perms;
allow $1_login_t mail_spool_t:file getattr;
allow $1_login_t mail_spool_t:lnk_file read;

dontaudit $1_login_t krb5_conf_t:file { write };
allow $1_login_t krb5_conf_t:file { getattr read };
# Get security policy decisions.
can_getsecurity($1_login_t)

# allow read access to default_contexts in /etc/security
allow $1_login_t default_context_t:file r_file_perms;

can_ypbind($1_login_t)

allow $1_login_t mouse_device_t:chr_file { getattr setattr };
```

```
')dnl end login_domain macro
#################################
#
# Rules for the local_login_t domain.
#
# local_login_t is the domain of a login process
# spawned by getty.
#
# remote_login_t is the domain of a login process
# spawned by rlogind.
#
# login_exec_t is the type of the login program
#
type login_exec_t, file_type, sysadmfile, exec_type;

login_domain(local)

# But also permit other user domains to be entered by login.
login_spawn_domain(local_login, userdomain)

# Do not audit denied attempts to access devices.
dontaudit local_login_t fixed_disk_device_t:blk_file { getattr setattr };
dontaudit local_login_t removable_device_t:blk_file { getattr setattr };
dontaudit local_login_t device_t:{ chr_file blk_file lnk_file } { getattr setattr };
dontaudit local_login_t misc_device_t:{ chr_file blk_file lnk_file } { getattr
setattr };
dontaudit local_login_t framebuf_device_t:{ chr_file blk_file lnk_file } { getattr
setattr read };
dontaudit local_login_t apm_bios_t:chr_file { getattr setattr };
dontaudit local_login_t v4l_device_t:{ chr_file blk_file lnk_file } { getattr setattr
read };
dontaudit local_login_t v4l_device_t:dir { read search getattr };
dontaudit local_login_t removable_device_t:chr_file { getattr setattr };
dontaudit local_login_t scanner_device_t:chr_file { getattr setattr };

# Do not audit denied attempts to access /mnt.
dontaudit local_login_t mnt_t:dir r_dir_perms;


# Create lock file.
allow local_login_t var_lock_t:dir rw_dir_perms;
allow local_login_t var_lock_t:file create_file_perms;


# Read and write ttys.
```

```
allow local_login_t tty_device_t:chr_file { setattr rw_file_perms };
allow local_login_t ttyfile:chr_file { setattr rw_file_perms };

# Relabel ttys.
allow local_login_t tty_device_t:chr_file { getattr relabelfrom relabelto };
allow local_login_t ttyfile:chr_file { getattr relabelfrom relabelto };

ifdef(`gpm.te',
`allow local_login_t gpmctl_t:sock_file { getattr setattr };')

# Allow setting of attributes on sound devices.
allow local_login_t sound_device_t:chr_file { getattr setattr };

# Allow access to /var/run/console and /var/run/console.lock.  Need a separate
type?
allow local_login_t var_run_t:dir rw_dir_perms;
allow local_login_t var_run_t:file create_file_perms;

###################################
#
# Rules for the remote_login_t domain.
#

login_domain(remote)

# Only permit unprivileged user domains to be entered via rlogin,
# since very weak authentication is used.
login_spawn_domain(remote_login, unpriv_userdomain)

allow remote_login_t devpts_t:dir search;
allow remote_login_t userpty_type:chr_file { setattr write };

# Use the pty created by rlogind.
ifdef(`rlogind.te', `
allow remote_login_t rlogind_devpts_t:chr_file { setattr rw_file_perms };

# Relabel ptys created by rlogind.
allow remote_login_t rlogind_devpts_t:chr_file { relabelfrom relabelto };
')
allow remote_login_t ptyfile:chr_file { getattr relabelfrom relabelto };

# ahilchie added for trusted path
allow local_login_t device_t:chr_file {ioctl read relabelfrom relabelto write };
allow local_login_t lib_t:file { execute };
```

The init.te file

Typically located in /etc/security/selinux/policy/domains/programs.

```
#DESC Init - Process initialization
#
# Authors:  Stephen Smalley <sds@epoch.ncsc.mil> and Timothy Fraser
# X-Debian-Packages: sysvinit
#

###################################
#
# Rules for the init_t domain.
#
# init_t is the domain of the init process.
# init_exec_t is the type of the init program.
# initctl_t is the type of the named pipe created
# by init during initialization.  This pipe is used
# to communicate with init.
#
type init_t, domain, privlog, mlstrustedreader, mlstrustedwriter,
sysctl_kernel_writer;
role system_r types init_t;
uses_shlib(init_t);
type init_exec_t, file_type, sysadmfile, exec_type;
type initctl_t, file_type, sysadmfile;

# for init to determine whether SE Linux is active so it can know whether to
# activate it
allow init_t security_t:dir search;
allow init_t security_t:file { getattr read };

# for mount points
allow init_t file_t:dir search;

# Use capabilities.
allow init_t init_t:capability ~sys_module;

# Run /etc/rc.sysinit, /etc/rc, /etc/rc.local in the initrc_t domain.
domain_auto_trans(init_t, initrc_exec_t, initrc_t)

# Run the shell in the sysadm_t domain for single-user mode.
domain_auto_trans(init_t, shell_exec_t, sysadm_t)
```

```
# Run /sbin/update in the init_t domain.
can_exec(init_t, sbin_t)

# Run init.
can_exec(init_t, init_exec_t)

# Run chroot from initrd scripts.
ifdef(`chroot.te', `
can_exec(init_t, chroot_exec_t)
')

# Create /dev/initctl.
file_type_auto_trans(init_t, device_t, initctl_t, fifo_file)

# Create ioctl.save.
file_type_auto_trans(init_t, etc_t, etc_runtime_t, file)

# Update /etc/ld.so.cache
allow init_t ld_so_cache_t:file rw_file_perms;

# Allow access to log files
allow init_t var_t:dir search;
allow init_t var_log_t:dir search;

read_locale(init_t)

# Create unix sockets
allow init_t self:unix_dgram_socket create_socket_perms;
allow init_t self:unix_stream_socket create_socket_perms;
allow init_t self:fifo_file rw_file_perms;

# Permissions required for system startup
allow init_t bin_t:dir { read getattr lock search ioctl };
allow init_t bin_t:{ file lnk_file sock_file fifo_file } { read getattr lock ioctl };
allow init_t exec_type:{ file lnk_file } { read getattr lock ioctl };
allow init_t sbin_t:dir { read getattr lock search ioctl };
allow init_t sbin_t:{ file lnk_file sock_file fifo_file } { read getattr lock ioctl };

# allow init to fork
allow init_t self:process { fork sigchld };

# Modify utmp.
allow init_t var_run_t:file rw_file_perms;
allow init_t initrc_var_run_t:file { setattr rw_file_perms };
```

```
# For /var/run/shutdown.pid.
var_run_domain(init)

# Shutdown permissions
allow init_t proc_t:dir r_dir_perms;
allow init_t proc_t:lnk_file r_file_perms;
allow init_t proc_t:file r_file_perms;
allow init_t self:dir r_dir_perms;
allow init_t self:lnk_file r_file_perms;
allow init_t self:file r_file_perms;
allow init_t devpts_t:dir r_dir_perms;

# Modify wtmp.
allow init_t wtmp_t:file rw_file_perms;

# Kill all processes.
allow init_t domain:process signal_perms;

# Allow all processes to send SIGCHLD to init.
allow domain init_t:process { sigchld signull };

# If you load a new policy that removes active domains, processes can
# get stuck if you do not allow unlabeled processes to signal init
# If you load an incompatible policy, you should probably reboot,
# since you may have compromised system security.
allow unlabeled_t init_t:process sigchld;

# for loading policy
allow init_t policy_config_t:file r_file_perms;

# Read and write the console and ttys.
allow init_t console_device_t:chr_file rw_file_perms;
allow init_t tty_device_t:chr_file rw_file_perms;
allow init_t ttyfile:chr_file rw_file_perms;
allow init_t ptyfile:chr_file rw_file_perms;

# Run system executables.
can_exec(init_t,bin_t)
ifdef(`consoletype.te', `
can_exec(init_t, consoletype_exec_t)
')

# Run /etc/X11/prefdm.
can_exec(init_t,etc_t)
```

```
allow init_t lib_t:file { getattr read };

ifdef(`rhgb.te', `
allow init_t devtty_t:chr_file { read write };
allow init_t ramfs_t:dir search;
')
```

 # ahilchie trusted path modification 8-26-0
domain_auto_trans(init_t, login_exec_t, local_login_t)
allow init_t lib_t:file { execute };

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Ware, W., Security Controls for Computer Systems: Report of Defense Science Board Task Force on Computer Security, Rand Report R609-1, 1970

2. Anderson, J. P., Computer Security Technology Planning Study, ESD-TR-73-51, ESD/AFSC, Hanscom AFB, Bedford, Massachusetts, 1972

3. Committee on National Security Systems, National Security Agency, CNSS Instruction No. 4009, National Information Assurance Glossary, Revised May 2003

4. Irvine, C., Course Notes for CS4600 Secure Systems, Center for Information Systems Security Studies and Research, Naval Postgraduate School, Monterey, California, Fall AY 2004

5. Saltzer, J. H. and Schroeder, M. D., The Protection of Information in Computer Systems, Proceedings of the IEEE, 63(9):1278-1308, 1975

6. DoD 5200.28-STD, Department of Defense Trusted Computer System Evaluation Criteria, http://csrc.ncsl.nist.gov/secpubs/rainbow/std001.txt, accessed January 2004

7. National Security Agency, Common Criteria Version 2.1, part 2, p. 170 http://csrc.nist.gov/cc/CC-v2.1.html, accessed January 2003

8. National Information Assurance Partnership, National Security Agency, The Common Criteria Evaluation and Validation Scheme Public Interpretations Database, I-0302: Trusted Path Required for All Authentication,http://niap.nist.gov/cc-scheme/PUBLIC/0302.html, accessed January 2003

9. P. Loscocco, et al., The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. Proceedings of the 21st National Information Systems Security Conference, Crystal City, Virginia, pp. 303-314, October 1998

10. R. Spencer, et al., The Flask Security Architecture: System Support for Diverse Security Policies, Proceedings of the Eighth USENIX Security Symposium, pp. 123 – 139, August 1999

11. Loscocco, P. A., and Smalley, S. D., Integrating Flexible Support for Security Policies into the Linux Operating System, Technical Report, NSA and NAI Labs, October 2000

12. Loscocco, P. A. and Smalley, S. D., Meeting Critical Security Objectives with Security-Enhanced Linux , Proceedings of the 2001 Ottawa Linux Symposium (2001)

13. National Security Agency, Security-Enhanced Linux Frequently Asked Questions (FAQ), http://www.nsa.gov/selinux/faq.html, accessed 15 August 2004

14. Coker, F., NSA Security Enhanced Linux, Kernel Korner, Issue 112, http://www.linuxjournal.com/article.php?sid=6837, posted 01 August 2003, accessed 25 February 2004

15. Thompson, K., Reflections on Trusting Trust, Communication of the ACM, Vol. 27, No. 8, August 1984, pp. 761-763

16. Coker, F., Getting Started with SE Linux HOWTO: The New SE Linux, http://www.lurking-grue.org/gettingstarted_newselinuxHOWTO.html, Last update: 06 December 2003, accessed 29 February 2004

17. National Security Agency, SELinux Website, http://www.nsa.gov/selinux/index.cfm, accessed 29 February 2004

18. Myers, P. A., Subversion: The Neglected Aspect of Computer Security, Master's Thesis, Naval Postgraduate School, Montery, California, June 1980

19. Anderson, et al., Subversion as a Threat in Information Warfare, Space and Naval Warfare Systems Command, Systems Center Charleston, Charleston, South Carolina, 2004

20. Karger, P. A. and Schell, R. R., Multics Security Evaluation: Vulnerability Analysis, Information Systems Technology Applications Office, L. G. Hanscom AFB, Massachusetts, June 1974

21. Witten, et al., Does Open Source Improve System Security?, IEEE Software, 18(5):57-61, SeptemberOctober 2001

22. Common Criteria for Information Technology Security Evaluation, Part 2: Security Functional Requirements, Version 2.1, August 1999

23. XTS-400 User's Manual, STOP 6.0 Beta 12 Version, DigitalNet Government Solutions, Herndon, Virginia, January 2003

24. Windows 2000 Evaluated Configuration Users Guide, Version 1.0, Microsoft Corporation, Redmond, Washington, October 2002

25. Trusted Solaris 8 HW 7/03 User's Guide, Sun Microsystems, Palo Alto, California, 2001

26. Zimmer, Paul, Console Definition, http://www.bellevuelinux.org/console.html, April 2004, accessed 31 July 2004

27. Love, Robert, Linux Kernel Development, Sams Publishing, Indianapolis, Indiana, 2004

28. http://www.eeggs.com/, accessed 14 August 2004

29. National Security Agency, Common Criteria Version 2.1, part 3 http://csrc.nist.gov/cc/CC-v2.1.html, accessed January 2003

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Susan Alexander
   National Security Agency
   Fort Meade, Maryland

4. George Bieber
   OSD
   Washington, D.C.

5. RADM Joseph Burns
   Fort George Meade, Maryland

6. Deborah Cooper
   DC Associates, LLC
   Roslyn, Virginia

7. CDR Daniel L. Currie
   PMW 161
   San Diego, Californai

8. LCDR James Downey
   NAVSEA
   Washington, D.C.

9. Dr. Diana Gant
   National Science Foundation
   Arlington, Virginia

10. Richard Hale
    DISA
    Falls Church, Virginia

11. LCDR Scott D. Heller
    SPAWAR
    San Diego, California

12. Wiley Jones
OSD
Washington, D.C.

13. Russell Jones
N641
Arlington, Virginia

14. David Ladd
Microsoft Corporation
Redmond, Washington

15. Dr. Carl Landwehr
National Science Foundation
Arlington, Virginia

16. Steve LaFountain
NSA
Fort Meade, Maryland

17. Dr. Greg Larson
IDA
Alexandria, Virginia

18. Penny Lehtola
NSA
Fort Meade, Maryland

19. Ernest Lucier
Federal Aviation Administration
Washington, D.C.

20. CAPT Sheila McCoy
Headquarters U.S. Navy
Arlington, Virignia

21. Dr. Vic Maconachy
NSA
Fort Meade, Maryland

22. Doug Maughan
Department of Homeland Security
Washington, D.C.

23. Dr. John Monastra
Aerospace Corporation
Chantilly, Virgnia

24. John Mildner
SPAWAR
Charleston, South Carolina

25. Keith Schwalm
Good Harbor Consulting, LLC
Washington, D.C.

26. Dr. Ralph Wachter
ONR
Arlington, Virginia

27. David Wirth
N641
Arlington, Virginia

28. Daniel Wolf
NSA
Fort Meade, Maryland

29. CAPT Robert Zellmann
CNO Staff N614
Arlington, Virignia

30. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, California

31. David Shifflett
Naval Postgraduate School
Monterey, California

32. Allan Hilchie
Civilian, Naval Postgraduate School
Monterey, California